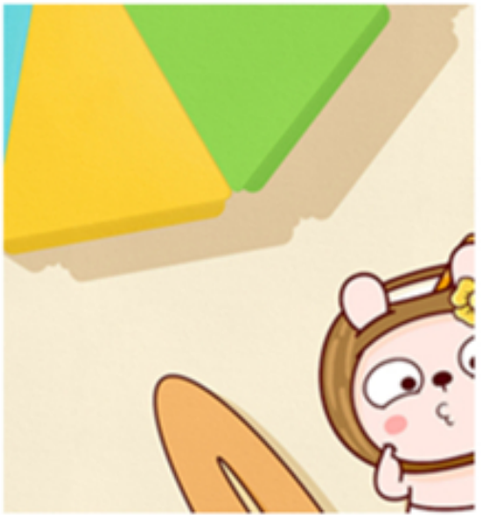
最新IT书籍PDF
最新IT书籍PDF
最新IT书籍PDF
最新IT书籍PDF



版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

非卖品，仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

大数据技术丛书

Data Lake for Enterprises

版权所有，严禁传播，违者自负法律责任！

非卖品，仅供非商业用途或交流学习使用

企业数据湖

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

张世武 李想 张浩林 译

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

版权所有，严禁传播，违者自负法律责任！

非卖品，仅供非商业用途或交流学习使用

严禁网络传播本PDF，违者责任自负！

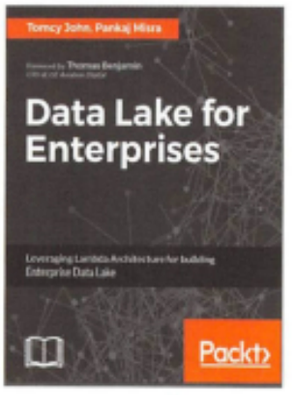
版权所有，严禁传播，违者自负法律责任！

非卖品，仅供非商业用途或交流学习使用

内容简介

“数据湖”已经成为大数据行业的一个重要术语，它是数据科学家们获得有意义的洞察力的平台，这些洞察力可以被企业用来重新定义或改变它们的运营方式。Lambda架构作为大数据领域的杰出模式之一也应运而生。

本书旨在帮助你选择正确的大数据技术并使用Lambda架构模式来为企业构建自己的数据湖。全书主要分为三部分，第一部分介绍数据湖的概念、数据湖在企业中的重要性以及Lambda架构。第二部分深入研究使用Lambda架构构建数据湖所需的主要技术组件，介绍流行的大数据技术，如Apache Hadoop、Sqoop、Flume、Flink和Elasticsearch等。第三部分展示一些非常实用的数据湖落地建议，包括企业应如何建立一个真实的数据湖，以及提高数据湖运营效率所需的其他技术组件。第三部分最后还详细介绍了几个实际的用例，以及在企业内实现这些用例时应考虑的重要方面。



原书封面

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

技术丛书

Data Lake for Enterprises

企业数据湖

[印度] 汤姆斯·约翰（Tomcy John） 潘卡·米斯拉（Pankaj Misra） 著
张世武 李想 张浩林 译

机械工业出版社
China Machine Press

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

图书在版编目（CIP）数据

企业数据湖 /（印）汤姆斯·约翰（Tomcy John），（印）潘卡·米斯拉（Pankaj Misra）著；张世武，李想，张浩林译 . —北京：机械工业出版社，2018.11
（大数据技术丛书）
书名原文：Data Lake for Enterprises

ISBN 978-7-111-61553-8

I. 企… II. ①汤… ②潘… ③张… ④李… ⑤张… III. 企业管理－数据管理－研究 IV. F272.7

中国版本图书馆 CIP 数据核字（2018）第 281948 号

本书版权登记号：图字 01-2017-7505

Tomcy John, Pankaj Misra: *Data Lake for Enterprises* (ISBN: 978-1-78728-134-9).

Copyright © 2017 Packt Publishing. First published in the English language under the title "*Data Lake for Enterprises*".

All rights reserved.

Chinese simplified language edition published by China Machine Press.

Copyright © 2019 by China Machine Press.

本书中文简体字版由 Packt Publishing 授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

企业数据湖

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：赵亮宇 责任校对：殷 虹

印 刷：北京市荣盛彩色印刷有限公司 版 次：2019 年 1 月第 1 版第 1 次印刷

开 本：186mm × 240mm 1/16 印 张：24.5

书 号：ISBN 978-7-111-61553-8 定 价：99.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88379426 88361066 投稿热线：（010）88379604

购书热线：（010）68326294 88379649 68995259 读者信箱：hzit@hzbook.com

版权所有 · 侵权必究

封底无防伪标均为盗版

本书法律顾问：北京大成律师事务所 韩光 / 邹晓东

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

The Translator's Words 译 者 序

大数据的概念最早可追溯到 20 世纪八九十年代，而 Google 发表的 3 篇论文（涉及 GFS、MapReduc 和 BigTable）则被认为是真正划时代的大数据学术文献。在工业界，Hadoop 及其生态圈则是大数据技术真正的标准。Hadoop 的理论基础源自 Google 发表的那 3 篇论文。经过十几年的蓬勃发展，大数据技术出现了前所未有的繁荣景象，开源社区及商业公司创造了覆盖大数据处理所有环节的多种技术框架和组件，企业和开发者可任意组合使用这些框架和组件来构建复杂的大数据解决方案。这也给使用者带来了极大的挑战——如何有效地使用这些庞杂的技术框架？是否存在一套完备的大数据理论框架？

Lambda 架构及数据湖（Data Lake）的概念应运而生，为使用者提供了一套切实可行的、功能完备的大数据解决方案。数据湖是一套非常灵活、松散耦合的架构方法，它将大数据处理划分为数据获取层、消息层、数据摄取层、数据存储层、Lambda 层（服务层、批处理层、快速处理层）等模块。这些模块之间既相互独立，又相互关联。在每个模块中可根据业务需求选择最适合的技术框架。数据湖能有效处理复杂的多源异构数据。Lambda 架构则是数据湖的灵魂，它为响应客户端的批处理、快速处理请求提供了一致的接口。总而言之，数据湖及 Lambda 架构能完美满足大数据 4V（Variety，Velocity，Volume，Veracity）的弹性要求，越来越受企业的青睐。

本书对数据湖与 Lambda 架构原理及实践做了极为精彩的阐述。全书分三部分，第一部分介绍了与大数据、数据湖及其重要组件相关的各种概念。第二部分逐章介绍了数据湖各模块涉及的技术框架，对每种框架的优缺点、架构原理、底层实现细节做了全面、深入的介绍。第三部分整合了第一部分和第二部分介绍的技术组件，为读者提供了数据湖的整体视图。本书内容深入浅出，图文并茂，辅以丰富的实操范例，是学习数据湖技术的绝佳读本。本书译文经过精心组织，结合了译者多年的大数据实践经验，并参考了腾讯、阿里巴巴、百度、今日头条、滴滴出行、东软等众多知名企业的业界专业人士的意见，经过多轮审校，力求信、

仅供非商业用途或交流学习使用

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持

编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

非卖品，仅供非商业用途或交流学习使用

达、雅。本书翻译团队由拥有丰富大数据项目实践经验的技术专家、架构师、管理者组成。由于数据湖与 Lambda 架构是最近出现的概念，业界尚无统一术语，再加上译者水平有限，难免存在一些翻译方面的问题，欢迎广大读者及同行批评指正。本书的第 1、3、6、7、9 章由李想完成，第 2、4、5、8、10 章由张世武完成，第 11 章和第 12 章由张浩林完成。张世武负责全书统稿。

翻译工作枯燥而冗长，并且占用了小伙伴们宝贵的业余时间，如果没有极大的热情，很难坚持到最后。这里还要感谢翻译团队家属的理解和支持，他们提供了宽松的家庭环境，让我们心无旁骛地投入本书的翻译工作中，并保质保量完成本书的翻译任务。

本书的翻译过程中，得到译者张世武、李想、张浩林所在单位——成都数联铭品科技有限公司（BBD）的大力支持。BBD 是一家行业领先的金融大数据公司，大数据及数据湖技术在 BBD 得到了广泛的应用和深入的实践。感谢曾途、尹康、丁国栋、田志伟、马扬、夏阳等领导的关心与支持，感谢业务部门及研发平台的小伙伴们为本书提出了宝贵意见和建议，尤其感谢数据平台的王尧同学。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

Forword 推 荐 序

在过去的四五十年里，企业的组织形态发生了翻天覆地的变化，在企业漫长而又稳健的演化过程中，它们发现了各种通过在经营领域增加 IT/ 软件系统来改进运营的途径和方法。所以，在世界财富 200 强公司中发现超过 250 个应用程序就不足为奇了。当企业成熟度发展到另一个阶段时，也会慢慢导致另外一个问题：系统之间互为孤岛，彼此之间很难交互。

随着企业的优化目标从局部优化转向全局优化，它们一直在利用一些新技术，如大数据技术，以找到方法和手段将数据从不同的 IT 系统汇集到一起，并进行进一步融合，以找到更好的方式来提高企业经营效率和有效性。这些方面的改进可以帮助企业节约成本。

Tomcy 和 Pankaj 在多个业务与技术领域有着丰富的经验，他们一直致力于寻找更好的方法来融合组织内各种应用程序产生的信息。他们经历了充满挑战和变数的技术、文化之旅。

本书从软件工程师、架构师和管理者的角度出发，极具实践价值，因为他们都参与、主导过各种各样的企业级系统实现，为企业贡献了价值。

使用未来证明模式和当代技术概念，如数据湖，能帮助企业为未来做好准备，甚至赋予企业整合之前的多个信息孤岛中的数据，并从中导出商业智能的能力。传统企业往往缺乏这方面的能力。

Thomas Benjamin

GE Aviation Digital 公司 CTO

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

关于作者 *About the Authors*

Tomcy John 生活在迪拜（阿联酋），出生于印度的喀拉拉邦，是一名企业级 Java 技术专家，拥有工学学士学位，并且有超过 14 年多行业的开发经验。他目前担任 Emirates Group IT 核心架构部门的首席架构师。在此之前，他曾在甲骨文公司、安永公司任职。擅长构建企业级应用。此外，他还在企业内担任首席导师和布道者，以促进新技术融入企业标准技术栈。工作之外，Tomcy 与年轻的开发者和工程师积极互动，并在各种论坛上作为技术布道者就许多话题发表自己的观点，从 Web 和中间件到各种持久性存储。可以在他的个人博客和 www.javacode book.com 网站上了解这些话题。

这是一个梦想成真的过程。我想把这本书献给我的父亲（Appachan），已故的 C.O.John，我最亲爱的妈妈（Ammachi），以及 Leela John，感谢你们帮助我取得今天的成绩。我还想借此机会，感谢我最亲爱的妻子 Serene 和两个可爱的孩子——Neil（儿子）、Anaya（女儿），感谢你们对我的支持与宽容，我一直沉醉于本书的撰写而疏于陪伴你们。

与合著者 Pankaj 合作是我的荣幸。借此机会感谢他对我的支持，从我动念头想撰写一本数据湖主题的著作开始，到完成这本书的所有阶段中，他一如既往地支持我。在我的职业生涯中，如果没有我的导师，我就不可能取得现在的成就。我要感谢 Thomas Benjamin（GE Aviation Digital CTO），Rajesh R.V（Emirates Group IT 首席架构师）和 Martin Campbell（首席架构师）在不同阶段支持我，用他们真诚的鼓励和丰富的知识来助我前行。

Pankaj Misra 是一名技术传播者，拥有工程学士学位，并且有超过 16 年跨多个业务领域的技术经验。自 2015 年以来，他一直在 Emirates Group IT 任职，之前曾在多个企业或组织任职。他擅长架构和构建基于多技术栈的解决方案及实现。他在印度的技术论坛上也是一名活

跃分子，参与过多个支持海量数据、可水平扩展的近实时数据处理和分析产品的构建。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

这本书的撰写对我来说是一个很好的机会，它将永远是我与合著者 Tomcy 合作和知识共享的一个纪念。非常感谢他对我的信任，并一直支持我。我想把这本书献给我的父亲 B. Misra 和我的母亲 Geeta Misra，他们对我有着最特殊的意义。同样地，我也非常感谢我的妻子 Priti 和我的孩子们——女儿 Eva 和儿子 Siddhant，感谢他们的理解、支持，并竭尽所能帮助我完成这本书。

书本是一种载体，用来回馈多年来在我职业生涯中传授给我知识的人们。我要感谢 Rajesh R.V.（Emirates Group IT 首席架构师）和 Thomas Benjamin（GE Aviation Digital CTO），他们始终在激励、帮助和支持着我们。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

关于技术审核人员 *About the Reviewers*

Wei Di 目前是一家商业分析数据挖掘团队的工作人员。作为一名数据科学家，她热衷于创建智能化和可扩展的分析、数据挖掘解决方案，这些解决方案可以影响数百万人，并为成功的商业行为赋能。

她的研究兴趣比较广泛，包括人工智能、机器学习和计算机视觉等领域。她就职于 eBay 人类语言技术团队和 eBay 研究实验室，专注于基于图像理解的大规模应用及视觉、文本信息的联合学习。在此之前，她就职于 Ancestry.com，从事记录链接、搜索相关性和排序相关的大规模数据挖掘及机器学习模型的工作。2011 年，她获得了普渡大学的博士学位，专注于数据挖掘和图像分类的研究。

Vivek Mishra 是一名 IT 从业人员，拥有超过 9 年的工作经验，精通 Java、J2EE、Hibernate、SCA4J、Mule、Spring、Cassandra、HBase、MongoDB、REDIS、Hive、Hadoop 等各种技术。他一直是开源软件的贡献者，如 Apache Cassandra，同时，他也是 Kundera 的核心贡献者（Kundera 是一个兼容 JPA 2.0 的对象数据存储映射库，用于 NoSQL 数据存储，如 Cassandra、HBase、MongoDB 和 REDIS）。

在他的职业生涯中，Vivek 与 SCM、银行和金融等行业中的知名公司有着持久的合作关系，在这些工作中使用了如敏捷、Scrum 等标准化的、全软件生命周期的方法。他目前受聘于 Impetus Infotech。

他曾在 cloud camp 和 Nasscom 等大型数据研讨会上发表演讲，并且是 mevivs.wordpress.com 上的活跃博主。

Rubén Oliva Ramos 是一名计算机系统工程师，拥有计算机和电子系统工程硕士学位，攻读的是电子信息学和网络专业，毕业于墨西哥瓜纳华托州莱昂市的 Salle Bajio 大学。他拥有超过 5 年的 Web 应用程序开发经验，擅长使用 Web 框架和云服务来构建物联网应用，以控制和监测与 Arduino 和 Raspberry Pi 连接的设备。

仅供非商业用途或交流学习使用

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

非卖品，仅供非商业用途或交流学习使用

同时，他还是 Salle Bajio 大学的机电学教师，负责教授硕士生的机电系统设计和工程课程。他还在莱昂市的 Centro de Bachillerato Tecnologico Industrial 225 工作，在 Mechatronics 职业技术学院教授以下课程：电子、机器人与控制、自动化和微控制器。他也是一名技术顾问，擅长 Android、iOS、Windows Phone、Visual Studio.NET、HTML5、PHP、CSS、Ajax、JavaScript、Angular、ASP.NET 数据库（SQLite、mongoDB 和 MySQL）和 Web 服务器（Node.js、IIS）等技术。Rubén 在 Arduino、Raspberry Pi、以太网防护、GPS 和 GSM/GPRS、ESP8266 等硬件编程领域也有涉猎，开发过数据采集类的控制监测系统，著有《Internet of Things Programming with JavaScript》一书。

Rubén 目前在 Alfaomega Editor Group 的工作内容包括用 Arduino 和 Visual Basic.NET 来监测、控制和获取数据。

感谢我的妻子 Mayte，我的儿子 Ruben 和 Dario，以及我的父母兄弟姐妹及其他家人。没有他们的支持，评审过程不会这么顺利。”

前　言 *Preface*

对许多企业来说，数据变得越来越重要，数据的重要性在多个方面得到了凸显。事实上，企业正在用核心数据来改善自己的运营。本书将首先介绍什么是数据，它与企业的相关性，以及企业如何利用这些数据以数字化的方式改变自己。为了利用数据，企业需要能存储海量数据的“仓库”，在当今时代，这些仓库不再被称为“数据仓库”，相反，它们被称为“数据湖”。

正如今天所看到的，我们有很多利用了大数据技术的用例。数据湖的概念已经存在了很长时间，但直到最近它才在企业中得到真正的关注。本书将这两个方面结合在一起，使用最

新的大数据技术，并遵循良好的架构模式，提供了可立即上手的、成熟的、能稳定运行的数据湖。

本书将把数据湖和 Lambda 架构结合在一起，帮助读者在他们的企业中实际操作这些内容。本书从较高的层次引入了大量的大数据技术，但是我们不打算将它定位为涉及很多技术框架的权威指南，因为书中涉及的每一种技术框架都值得用一本书来深入介绍它们。

本书用所选的技术框架涵盖了数据湖、Lambda 架构模式的解释和实现。当然，这些具体的技术可以在将来或者根据组织设置的标准来替换。所以，本书内容中的精髓不仅在现在，而且在将来很长一段时间内都是有效的。因此，与同一领域的其他那些依赖具体技术框架的书籍相比，本书的“保鲜期”更长。

本书将带你踏上一段奇妙的旅程，在这个过程中，它遵循一种既直观又令人兴奋的结构。

本书涵盖的内容

本书分为 3 部分。每部分都包含若干章，当一个部分完成后，读者将会从整体上理解该部分内容。这些部分的设计和结构使得读者首先了解主要的功能和技术；然后在接下来的部

分，或者更确切地说，在本书最后一部分中，一切都会被整合在一起。在本书的末尾，读者将实现一个可操作的数据湖。

第一部分，向读者介绍了与数据、数据湖及其重要组件相关的各种概念。此部分共分 4 章，如下所列，各章都有明确的目标。

第 1 章，以通用的方式向读者介绍什么是数据，以及数据与企业的相关性。该章解释了在当前社会数据为什么重要，以及应该如何使用它们。现实生活中的各种用例已经展示了数据的重要性以及数据是如何改变企业的。这些真实的案例将启发读者的创造性思维，并思考如何利用数据来改变自己的企业。

第 2 章，进一步深入了解数据湖概念的细节，并解释使用数据湖能解决企业面临的哪些问题。该章还提供了关于 Lambda 架构以及如何将其用于数据湖的内容。到该章结束，读者将了解数据湖的概念和组织构建数据湖所采用的各种方法。

第 3 章，详细介绍 Lambda 架构、它的各种组件以及数据湖与此架构模式之间的关联。在该章中，读者将详细了解 Lambda 架构，包括它产生的原因和能解决的具体问题。该章还将帮助读者理解 Lambda 架构的核心概念以及如何在企业中应用它的能力。读者还将理解各种设计模式和组件，在构建 Lambda 架构时它们可以帮助实现批处理和实时数据处理。现在，读者对数据、数据湖和 Lambda 架构应该有了足够的了解，可以跳到下一节，学习如何为所在企业实现数据湖。

第 4 章，向读者介绍可用于 Lambda 架构每一层（组件）的技术，还将帮助读者选择一个市场中领先且让我们感觉非常棒的技术。在该章中，读者将了解当前大数据技术领域中的各

种 Hadoop 发行版本，以及如何利用它们在企业数据湖中构建 Lambda 架构。在这些技术背景下，读者将了解企业数据湖中批处理层、快速处理层和服务层背后的细节及设计动机。

第二部分，向读者介绍了许多技术，这些技术将是数据湖实现的一部分。此部分中的每一章都涉及一种技术框架，它们逐步被用于构建数据湖和用例，即单一客户视图（SCV）。几乎每一章中讨论的技术的所有重要技术细节都将以整体的方式被涵盖，因为过于深入的探讨超出了本书的范围。此部分由 6 章组成，每章都有一个明确的目标，如下所述。

第 5 章，对 Apache Sqoop 做了深入的介绍，给出了选择这种技术框架的理由，也向读者提供了其他技术选项相关的大量细节。该章还给出了数据湖和 Lambda 架构的详细实例。在该章中，读者将理解 Sqoop 框架以及其他用于将数据从企业数据源加载到数据湖的类似工具。读者也将了解 Sqoop 架构相关的各种技术细节，还可以通过示例了解 Sqoop 的实际操作以及使用 Hadoop 技术时涉及的各种步骤。

第 6 章，将深入学习 Apache Flume，它是一个中间环节，与数据湖及 Lambda 架构中的

版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播，违规者造成的法律责任和后果，违规者自负
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

其他技术框架衔接。读者将会理解 Flume 作为一个框架，数据湖通过哪些模式来使用它。读者还将了解 Flume 架构和技术实现细节，并使用该框架来获取和使用数据，其中还包括了使用特定的功能来处理事务控制、数据重放的示例。读者还将了解如何使用流式数据处理技术来进行基于流的处理。

第 7 章，将深入介绍 Apache Kafka。该章最开始给出了选择一种特定技术的理由，同时列举了其他技术选项的细节。在这一章中，读者将会理解 Kafka 是一个面向消息的中间件，同时会了解它与其他消息中间件的差异。读者可以从客户端（发布者）、代理和消费者（订阅者）的角度来了解 Kafka 及其功能的细节，以及如何利用它来构建水平扩展能力。读者还将了解如何将 Kafka 与 Hadoop 组件集成以获取企业数据，以及这种集成给数据湖带来的功能。

第 8 章，读者将理解基于流式数据和流式处理的概念，特别是关于 Apache Flink 的概念。读者将深入研究数据湖和大数据技术背景下的 Apache Flink，通过示例演示 Flink 的实时数据处理。读者还将认识到流式处理功能如何依赖于架构中的各种其他层，以及这些层如何影响流式处理能力。

第 9 章，将深入介绍 Apache Hadoop。在这一章中，读者将更深入地了解 Hadoop，对其有一个全面的认识，同时尝试使用各种 Hadoop 组件及一些特定功能，这些组件可以为企业数据湖提供服务。在数据湖背景下，将在实现级上对 Hadoop 进行解释，如 Hadoop 框架是如何实现文件存储、不同文件格式数据处理和 MapReduce 范式的。Hadoop 可以构成数据湖的基础，Hadoop 技术栈也能提供近实时处理能力。

第 10 章，会深入研究 Elasticsearch。读者将理解 Elasticsearch 作为数据索引框架的作用以及如何利用该框架提供的各种分析器进行有效的搜索。读者还将理解数据湖如何有效利用 Elasticsearch 的分片和分发机制来提供高效的一致性数据服务，了解如何使用 Elasticsearch 来处理快速流式数据，及构建高性能应用。

第三部分，将汇集本书第一部分和第二部分的技术组件，为读者提供数据湖的整体视图。我们将以简短的方式引入额外的概念和技术，如果有需要，读者可根据企业需求更深入地探索这些内容。另外，深入研究该章所涵盖的技术超出了本书的范围。但是，我们仍希望读者了解这些附加技术，以及在需要时如何将它们引入数据湖实现。本部分由两章组成，每章都有一个明确的目标，下面会详细叙述。

第 11 章，将所有技术点整合在一起，在向读者介绍完数据湖、Lambda 架构及其他各种技术之后，为读者展示了整体的图景。到了这个阶段，读者应该有成就感，这时可以将代码库组织起来，展示它是如何工作的。在这一章中，读者将会意识到如何集成数据湖的各个组

件来实现一个功能完备的数据湖。读者还将通过运行范例来实现数据湖的完整性，将前几章学习的内容结合在一起，完成一个可运行的数据湖实现。

第 12 章，贯穿本书，读者通过单一客户视图的形式来理解用例；然而，在浏览本书的过程中，还有其他与组织相关的用例，读者可对它们进行深入的思考。在这一章中，这种思想得到了进一步的阐发。读者将理解并实现各种用例，这些用例可以从数据湖中获得巨大的回报，并帮助优化它们的持有、运营和活动成本，还能从这些数据使用中推导出商业智能。在该章中，读者还将认识到这些用例的多样性以及企业数据湖能为这些用例提供多大程度的帮助。

阅读本书的必备条件

本书面向开发人员、架构师和产品 / 项目负责人，用于实现基于 Lambda 架构的企业数据湖。书中包括可运行范例，用以帮助读者理解和观察围绕数据湖及其基本实现的各种概念。为了运行这些范例，需要使用各种开源软件、所需的基础设施和开发 IDE。我们努力保持示例简洁并利用常见的框架和组件。用于运行这些示例的操作系统为 CentOS 7，但是这些示例可以在 Linux 操作系统的任何版本上运行。

本书的目标读者

- 致力于实现企业数据湖的 Java 开发者和架构师。
- 致力于实践 Lambda 架构和大数据技术的 Java 开发者。
- 致力于探索大数据技术并渴望利用大数据技术提供解决方案的 Java 开发者。

排版约定

在本书中，不同的文本样式用来区分不同的信息。下面是一些例子，以及每种样式的解释。

代码块样式如下所示：

```
agent.sources = spool-source
agent.sources.spool-source.type=spooldir
agent.sources.spool-source.spoolDir=/home/centos/flume-data
agent.sources.spool-source.interceptors=ts uuid
```

命令行输入或输出样式如下所示：

```
${FLUME_HOME}/bin/flume-ng agent --conf ${FLUME_HOME}/conf/  -f
${FLUME_HOME}/conf/spool-fileChannel-kafka-flume-conf.properties  -n
agent -Dflume.root.logger=INFO,console
```

警告或重要的注意事项。

提示或技巧。

范例下载

本书的范例代码可以从 http://www.packtpub.com 通过个人账号下载，也可以访问华章图书官网 http://www.hzbook.com，通过注册并登录个人账户下载。

本书涉及的代码也托管在 GitHub 上，请参考 https://github.com/PacktPublishing/Data-Lake-for-Enterprises。

Contents 目　录

译者序
推荐序
关于作者
关于技术审核人员
前言

第一部分　概述

第 1 章　数据导论 ····· 2

1.1　探索数据 ····· 3
1.2　什么是企业数据 ····· 4
1.3　企业数据管理 ····· 5
1.4　大数据相关概念 ····· 6
1.5　数据与企业的相关性 ····· 8
1.6　数据质量 ····· 9
1.7　企业中数据如何存放 ····· 10
1.7.1　内联网（企业内部）····· 10
1.7.2　互联网（企业外部）····· 10
1.7.3　数据持久化存储（RDBMS 或者 NoSQL）····· 12
1.7.4　传统的数据仓库 ····· 13
1.7.5　文件存储 ····· 13
1.8　企业现状 ····· 14
1.9　企业数字化转型 ····· 15
1.10　数据湖用例启示 ····· 16
1.11　总结 ····· 17

第 2 章　数据湖概念概览 ····· 18

2.1　什么是数据湖 ····· 18
2.2　数据湖如何帮助企业 ····· 19
2.3　数据湖是如何工作的 ····· 20
2.4　数据湖与数据仓库的区别 ····· 21
2.5　数据湖的构建方法 ····· 22
2.6　Lambda 架构驱动的数据湖 ····· 22
2.6.1　数据摄取层——摄取数据用于处理和存储 ····· 23
2.6.2　批处理层——批量处理已提取数据 ····· 23
2.6.3　快速处理层——近实时数据处理 ····· 24
2.6.4　数据存储层——存储所有数据 ····· 24
2.6.5　服务层——数据交付与导出 ····· 25
2.6.6　数据获取层——从源系统获取数据 ····· 25

2.6.7 消息层——数据传输的保障 … 26
2.6.8 探索数据摄取层 … 27
2.6.9 探索 Lambda 层 … 28
2.7 总结 … 35

第 3 章 Lambda 架构：一种数据湖实现模式 … 36

3.1 什么是 Lambda 架构 … 36
3.2 Lambda 架构简史 … 37
3.3 Lambda 架构的原则 … 37
3.3.1 容错原则 … 38
3.3.2 不可变数据原则 … 38
3.3.3 重新计算原则 … 38
3.4 Lambda 架构的组件 … 38
3.4.1 批处理层 … 39
3.4.2 快速处理层 … 41
3.4.3 服务层 … 43
3.5 Lambda 架构的完整工作原理 … 44
3.6 Lambda 架构的优势 … 45
3.7 Lambda 架构的劣势 … 46
3.8 Lambda 架构技术概览 … 46
3.9 应用 Lambda … 47
3.9.1 企业级日志分析 … 47
3.9.2 获取和分析传感器数据 … 47
3.9.3 电子邮件平台实时统计 … 48
3.9.4 实时赛事分析 … 48
3.9.5 推荐引擎 … 48
3.9.6 安全威胁分析 … 48
3.9.7 多渠道用户行为分析 … 48
3.10 Lambda 架构运行范例 … 48
3.11 Kappa 架构 … 49
3.12 总结 … 50

第 4 章 数据湖中的 Lambda 应用 … 51

4.1 Hadoop 发行版本介绍 … 51
4.2 影响企业大数据技术栈选择的因素 … 53
4.2.1 技术能力 … 53
4.2.2 是否易于部署和维护 … 53
4.2.3 集成准备 … 53
4.3 批处理层与数据处理 … 53
4.3.1 NameNode 服务器 … 54
4.3.2 Secondary NameNode 服务器 … 55
4.3.3 YARN … 55
4.3.4 数据存储节点 … 55
4.3.5 快速处理层 … 56
4.3.6 Flume 用于数据获取 … 57
4.3.7 Spark Streaming … 58
4.4 服务层 … 62
4.4.1 数据存储层 … 62
4.4.2 数据访问层 … 63
4.5 总结 … 64

第二部分 数据湖的技术组件

第 5 章 基于 Apache Sqoop 的批量数据获取 … 68

5.1 数据湖背景中的数据获取 … 68
5.1.1 数据获取层 … 68
5.1.2 批量数据获取——技术路线图 … 69
5.2 为什么使用 Apache Sqoop … 70

5.2.1 Sqoop 简史……71
5.2.2 Sqoop 的优势……71
5.2.3 Sqoop 的劣势……72
5.3 Sqoop 的功能……72
5.3.1 Sqoop 2 的架构……74
5.3.2 Sqoop 1 与 Sqoop 2……75
5.3.3 Sqoop 的功能……77
5.3.4 使用 Sqoop 导入数据……77
5.3.5 使用 Sqoop 导出数据……78
5.4 Sqoop connector……79
5.5 Sqoop 对 HDFS 的支持……81
5.6 Sqoop 运行范例……81
5.6.1 安装与配置……81
5.6.2 数据源配置……90
5.6.3 Sqoop 配置（数据库驱动）……91
5.6.4 将 HDFS 配置为目的地……91
5.6.5 Sqoop 数据导入……91
5.6.6 Sqoop 数据导出……97
5.6.7 Sqoop job……98
5.6.8 Sqoop 2……99
5.6.9 SCV 用例视角中的 Sqoop……102
5.7 适合使用 Sqoop 的场景……103
5.8 不适合使用 Sqoop 的场景……104
5.9 实时 Sqooping 是否可行……104
5.10 其他选项……104
5.10.1 原生大数据 connector……105
5.10.2 Talend……106
5.10.3 Pentaho Kettle（PDI——Pentaho 数据集成）……106
5.11 总结……106

第 6 章 基于 Apache Flume 的流式数据获取……108

6.1 数据获取……108
6.1.1 什么是流式数据……109
6.1.2 批量数据和流式数据……110
6.1.3 流式数据获取——技术路线图……110
6.1.4 什么是 Flume……111
6.1.5 Sqoop 和 Flume……112
6.2 为什么使用 Flume……113
6.2.1 Flume 简史……113
6.2.2 Flume 的优势……113
6.2.3 Flume 的劣势……114
6.3 Flume 的架构原则……114
6.4 Flume 架构……115
6.4.1 Flume 架构之一：分布式数据流水线……116
6.4.2 Flume 架构之二：扇出……117
6.4.3 Flume 架构之三：扇入……117
6.4.4 Flume 架构中的 3 层设计……118
6.4.5 高级 Flume 架构……118
6.4.6 Flume 的可靠性级别……120
6.5 Flume 事件——流式数据……120
6.6 Flume Agent……120
6.7 Flume Source……122
6.8 Flume Channel……123
6.9 Flume Sink……125
6.10 Flume 配置……126
6.11 Flume 事务管理……127
6.12 Flume 的其他组件……128
6.12.1 Channel Processor……128

非卖品，仅供非商业用途或交流学习使用

6.12.2 Interceptor ····· 129
6.12.3 Channel Selector ····· 129
6.12.4 Sink Group ····· 130
6.12.5 事件序列化 ····· 131
6.13 上下文路由 ····· 131
6.14 Flume 运行范例 ····· 132
6.14.1 安装和配置 ····· 132
6.14.2 SCV 用例中的 Flume ····· 133
6.15 适合使用 Flume 的场景 ····· 145
6.16 不适合使用 Flume 的场景 ····· 145
6.17 其他选项 ····· 146
6.17.1 Apache Flink ····· 146
6.17.2 Apache NiFi ····· 146
6.18 总结 ····· 147

第 7 章 使用 Apache Kafka 构建消息层 ····· 148

7.1 数据湖背景中的消息层 ····· 148
7.1.1 消息层 ····· 148
7.1.2 消息层——技术路线图 ····· 149
7.1.3 什么是 Apache Kafka ····· 150
7.2 为什么使用 Apache Kafka ····· 150
7.2.1 Kafka 简史 ····· 151
7.2.2 Kafka 的优势 ····· 152
7.2.3 Kafka 的劣势 ····· 153
7.3 Kafka 的架构 ····· 153
7.3.1 Kafka 架构的核心原则 ····· 153
7.3.2 数据流的生命周期 ····· 154
7.3.3 Kafka 的工作原理 ····· 155
7.3.4 Kafka 的消息 ····· 156
7.3.5 Kafka 生产者 ····· 157
7.3.6 Kafka topic 中的数据持久化 ····· 157
7.3.7 Kafka 中 topic 的划分：partition ····· 158
7.3.8 Kafka 中的消息 broker ····· 159
7.3.9 Kafka 的消费者 ····· 160
7.4 其他 Kafka 组件 ····· 161
7.4.1 ZooKeeper ····· 161
7.4.2 MirrorMaker ····· 161
7.5 Kafka 编程接口 ····· 162
7.5.1 Kafka 核心 API ····· 162
7.5.2 Kafka REST 接口 ····· 162
7.6 生产者和消费者的可靠性 ····· 162
7.7 Kafka 的安全性 ····· 163
7.8 Kafka 作为面向消息的中间件 ····· 164
7.9 Kafka 与水平可扩展架构 ····· 165
7.10 Kafka 连接 ····· 165
7.11 Kafka 运行范例 ····· 166
7.11.1 安装 ····· 166
7.11.2 生产者：向 Kafka 写入数据 ····· 167
7.11.3 消费者：从 Kafka 获取数据 ····· 171
7.11.4 设置多 broker 集群 ····· 173
7.11.5 SCV 用例中的 Kafka 应用 ····· 176
7.12 适合使用 Kafka 的场景 ····· 176
7.13 不合适使用 Kafka 的场景 ····· 177
7.14 其他选项 ····· 177
7.14.1 RabbitMQ ····· 177
7.14.2 ZeroMQ ····· 179
7.14.3 Apache ActiveMQ ····· 179
7.15 总结 ····· 180

非卖品，仅供非商业用途或交流学习使用

第 8 章 使用 Apache Flink 处理数据 …… 181

8.1 数据湖背景中的数据摄取层 …… 181

8.1.1 数据摄取层 …… 182

8.1.2 数据摄取层——技术路线图 …… 183

8.1.3 什么是 Apache Flink …… 184

8.2 为什么使用 Apache Flink …… 184

8.2.1 Flink 简史 …… 185

8.2.2 Flink 的优势 …… 186

8.2.3 Flink 的劣势 …… 187

8.3 Flink 的工作原理 …… 187

8.3.1 Flink 架构 …… 187

8.3.2 Flink 架构的核心原则 …… 192

8.3.3 Flink 组件栈 …… 192

8.3.4 Flink 中的 Checkpointing …… 192

8.3.5 Flink 中的 Savepoint …… 194

8.3.6 Flink 中的流窗口选项 …… 195

8.3.7 内存管理 …… 197

8.4 Flink 的 API …… 197

8.4.1 DataStream API …… 198

8.4.2 DataSet API …… 200

8.4.3 Flink 领域相关库 …… 202

8.5 Flink 运行范例 …… 203

8.5.1 安装 …… 204

8.5.2 范例——使用 Flink 处理数据 …… 205

8.5.3 SCV 用例中的 Flink …… 212

8.6 适合使用 Flink 的场景 …… 217

8.7 不适合使用 Flink 的场景 …… 218

8.8 其他选项 …… 218

8.8.1 Apache Spark …… 218

8.8.2 Apache Storm …… 219

8.8.3 Apache Tez …… 219

8.9 总结 …… 220

第 9 章 使用 Apache Hadoop 存储数据 …… 221

9.1 数据湖背景中的数据存储和 Lambda 批处理层 …… 221

9.1.1 数据存储和 Lambda 批处理层 …… 222

9.1.2 数据存储和 Lambda 批处理层——技术路线图 …… 223

9.1.3 什么是 Apache Hadoop …… 224

9.2 为什么使用 Hadoop …… 224

9.2.1 Hadoop 简史 …… 225

9.2.2 Hadoop 的优势 …… 225

9.2.3 Hadoop 的劣势 …… 226

9.3 Hadoop 的工作原理 …… 227

9.3.1 Hadoop 架构的核心原则 …… 227

9.3.2 Hadoop 架构 …… 228

9.3.3 Hadoop 架构组件 …… 231

9.3.4 Hadoop 架构细节 …… 233

9.4 Hadoop 生态系统 …… 234

9.4.1 数据访问 / 处理组件 …… 235

9.4.2 数据存储组件 …… 236

9.4.3 监控、管理和协调组件 …… 237

9.4.4 数据集成组件 …… 239

9.5 Hadoop 发行版 …… 240

9.6 HDFS 和数据格式 …… 241

9.7 Hadoop 用于近实时应用 …… 242

非卖品，仅供非商业用途或交流学习使用

9.8 Hadoop 部署模式 ····· 243
9.9 Hadoop 运行范例 ····· 243
9.9.1 安装 ····· 244
9.9.2 数据准备 ····· 244
9.9.3 安装 Hive ····· 244
9.9.4 范例——批量数据加载 ····· 247
9.9.5 范例——MapReduce 数据处理 ····· 248
9.9.6 SCV 用例中的 Hadoop ····· 254
9.10 不适合使用 Hadoop 的场景 ····· 262
9.11 其他选项 ····· 263
9.12 总结 ····· 263

第 10 章 使用 Elasticsearch 存储全文索引 ····· 264

10.1 数据湖背景中的数据存储层与 Lambda 快速处理层 ····· 264
10.1.1 数据存储层与 Lambda 快速处理层 ····· 265
10.1.2 数据存储层与 Lambda 快速处理层——技术路线图 ····· 265
10.2 什么是 Elasticsearch ····· 266
10.3 为什么使用 Elasticsearch ····· 266
10.3.1 Elasticsearch 简史 ····· 267
10.3.2 Elasticsearch 的优势 ····· 268
10.3.3 Elasticsearch 的劣势 ····· 269
10.4 Elasticsearch 的工作原理 ····· 269
10.4.1 Elasticsearch 架构的核心原则 ····· 269
10.4.2 Elasticsearch 术语 ····· 270
10.5 Elastic Stack ····· 273
10.5.1 Kibana ····· 273
10.5.2 Elasticsearch ····· 275
10.5.3 Logstash ····· 276
10.5.4 Beats ····· 277
10.6 Elastic Cloud ····· 279
10.7 Elasticsearch DSL ····· 281
10.8 Elasticsearch 中的节点 ····· 283
10.8.1 Master 节点 ····· 283
10.8.2 Data 节点 ····· 283
10.8.3 Client 节点 ····· 284
10.9 Elasticsearch 与关系数据库 ····· 284
10.10 Elasticsearch 生态系统 ····· 285
10.10.1 Elasticsearch 分析器 ····· 285
10.10.2 Elasticsearch 插件 ····· 286
10.11 Elasticsearch 部署选项 ····· 287
10.12 Elasticsearch Client ····· 287
10.13 Elasticsearch 用于快速流式处理 ····· 288
10.14 Elasticsearch 作为数据源 ····· 289
10.15 Elasticsearch 用于内容索引 ····· 289
10.16 Elasticsearch 与 Hadoop ····· 289
10.17 Elasticsearch 运行范例 ····· 290
10.17.1 安装 ····· 291
10.17.2 创建和删除索引 ····· 293
10.17.3 对文档进行索引 ····· 294
10.17.4 获取被索引的文档 ····· 296
10.17.5 搜索文档 ····· 296
10.17.6 更新文档 ····· 299
10.17.7 删除文档 ····· 299
10.17.8 SCV 用例中的 Elasticsearch ····· 300

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

10.18 适合使用 Elasticsearch 的场景 ····· 317
10.19 不适合使用 Elasticsearch 的场景 ····· 317
10.20 其他选项 ····· 318
10.21 总结 ····· 319

第三部分 将所有技术整合在一起

第 11 章 数据湖组件集成 ····· 322

11.1 数据湖的学习进程 ····· 322
11.2 数据湖架构的核心原则 ····· 324
11.3 企业数据湖面临的挑战 ····· 324
11.4 企业对数据湖的期望 ····· 326
11.5 数据湖的其他用途 ····· 326
11.6 了解更多关于数据存储的信息 ····· 327
11.6.1 数据存储区域 ····· 327
11.6.2 数据模式和模型 ····· 329
11.6.3 存储选项 ····· 329
11.6.4 压缩方法 ····· 331
11.6.5 数据分区 ····· 332
11.7 更多关于数据处理的知识 ····· 333
11.7.1 数据校验和清洗 ····· 333
11.7.2 机器学习 ····· 334
11.7.3 调度器 / 工作流 ····· 334
11.7.4 Apache Oozie ····· 335
11.7.5 复杂事件处理 ····· 340
11.8 数据安全 ····· 341
11.8.1 Apache Knox ····· 342
11.8.2 Apache Ranger ····· 342
11.8.3 Apache Sentry ····· 344
11.9 数据加密 ····· 345
11.10 元数据管理和治理 ····· 346
11.10.1 元数据 ····· 346
11.10.2 数据治理 ····· 347
11.10.3 数据世系 ····· 347
11.10.4 如何实现 ····· 348
11.11 数据审计 ····· 350
11.12 数据可追溯性 ····· 350
11.13 了解更多服务层细节 ····· 351
11.13.1 服务层构建原则 ····· 351
11.13.2 服务类型 ····· 351
11.13.3 服务层组件 ····· 353
11.13.4 数据导出 ····· 355
11.13.5 混合数据访问 ····· 355
11.13.6 范例——服务层 ····· 355
11.14 总结 ····· 360

第 12 章 数据湖用例建议 ····· 361

12.1 在企业中推行网络安全实践 ····· 361
12.2 深入了解企业的客户 ····· 362
12.3 提升仓储管理效率 ····· 363
12.4 品牌创建与企业营销 ····· 364
12.5 为客户提供更个性化的服务 ····· 366
12.6 让物联网数据触手可及 ····· 367
12.7 更实用的数据归档 ····· 367
12.8 现有的数据仓库基础设施 ····· 368
12.9 实现电信安全和法规遵从 ····· 368
12.10 总结 ····· 369

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

第一部分 Part 1

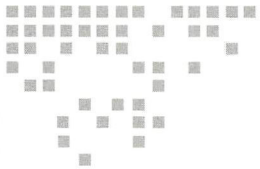
概述

此部分向读者介绍了与数据、数据湖及其重要组件相关的各种概念，共分 4 章，如下所示：

- 第 1 章　数据导论
- 第 2 章　数据湖概念概览
- 第 3 章　Lambda 架构：一种数据湖实现模式
- 第 4 章　数据湖中的 Lambda 应用

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用



Chapter 1 第 1 章

数据导论

本书中，我们将着手为企业实施一项技术杰作。在这个过程中，读者不仅要学习许多新的工具和技术，还要学习大量的术语和理论知识。这些会帮助你达到最终的目标，就是构建这个技术杰作——数据湖。

本章旨在让读者为前方艰难的道路做好准备，这样读者就可以清晰地知道将要达到的目标是什么。数据湖的概念随着时间的推移在企业中不断演变，最初起源于数据仓库的概念。数据仓库包含长期积累的数据，并根据报表需求和历史需求以不同的方式存储。然后，数据集市的概念出现了，它提供企业属性相关的小数据集。数据湖从上述这些概念演变而来，成为企业的中心数据库，可以捕获原始数据，生成加工后的数据，提供最关键的企业信息。

数据湖相关的话题或者说技术已经不新鲜了，但很少有企业实施了功能齐全的数据湖。通过本书，我们希望企业开始认真考虑投资建设数据湖。此外，希望在工程师的帮助下，可以让企业的最高管理层了解数据湖可以提供的功能和适用的场景。

所以，让我们开始这一段旅程吧。

请放心，读完本书后，读者将可以帮助企业（无论规模大或小）用以数据为中心的方式进行思考和对业务建模，并使用数据湖作为核心技术。

本章的目的是让读者深入了解数据、大数据以及数据相关的一些重要细节。本章给出了一些重要的基于教科书的定义，这些定义读者需要深刻理解，以便能明确数据与企业的关联。读者也将认识到数据与大数据之间的关键差异。本章将深入探讨不同类型的数据，以及这些数据分布在企业的哪些地方。

本章的后半部分将介绍企业在数据管理方面的现状，希望读者能得到启迪；同时试图

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

提供一个高层次的视角，来了解企业向以数据为核心进行转型的目标。整本书基于一个真实的用例，在1.10节会更加详细地介绍这个用例。读者可以通过这个用例掌握很多相关的概念。

1.1 探索数据

数据是指一组定性或定量变量的值。

数据可以被测量、收集、报告和分析，然后可以使用图形、图像和其他分析工具进行可视化。一般意义上，数据指的是现有的信息或知识，并以一种适合使用或处理的方式进行表示或编码。

——维基百科

数据大致分为3种类型：

- 结构化数据（structured data）
- 非结构化数据（unstructured data）
- 半结构化数据（semi-structured data）

结构化数据指的是我们通常在业务应用程序中获取的，在关系型数据库管理系统（Relational Database Management System，RDBMS）中或者以非关系型数据库（NoSQL最初称为non-SQL）的形式存储的数据。

结构化数据又可以大致分为两类，即原始数据（raw data）和清洁数据（cleansed data）。原始数据是指以原样获取的，没有经过清洗和过滤的数据。清洁数据是指经过大量清洗和过滤来适配业务用户的分析需求的数据。

所有其他不属于结构化数据类型的数据都称为非结构化数据。视频、图片等形式的数据都属于非结构化数据。

还有半结构化数据，它随着互联网的兴起而出现，并且随着社交网站的发展愈发成为主流。维基百科对半结构化数据的定义如下：

半结构化数据其实是结构化数据的一种形式，它不符合关系型数据库或者其他数据表格形式的数据模型所表示的规范结构，但是它包含标签或其他标记来分割语义元素，并要求在数据内保存记录和字段的层次结构。因此，也被称为自描述结构（self-describing structure）。

有一些众所周知的半结构化数据的例子，如JavaScript对象表示法（JSON）和可扩展标记语言（XML）。

图1-1中涵盖了我们讨论的各种数据类型。读者请不要因为在结构化数据部分看到电子表格和文本文件而疑惑。这是因为图1-1中的数据是以记录的形式表达的，确实属于结构化数据。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

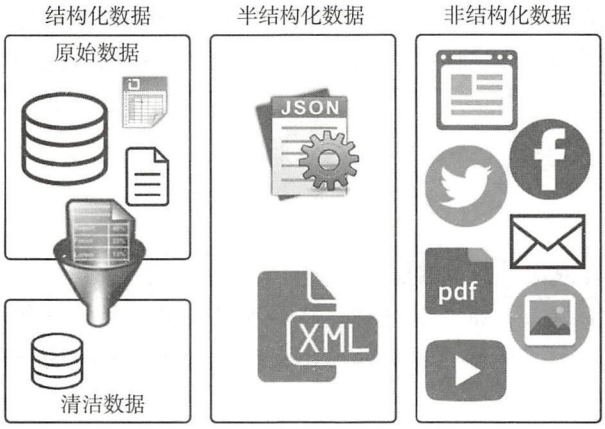


图 1-1 数据类型

1.2 什么是企业数据

企业数据指的是企业内部员工及其合作伙伴跨越不同部门、不同地点而共享，跨越不同大洲而传播的数据。这些数据对企业具有很高的价值，包括财务数据、业务数据、员工个人数据等，企业花费了大量时间和金钱来保证数据在各方面的安全和质量。

然而，所谓的企业数据从当前状态变得日渐陈旧，虽然以某种形式进行存储，但是难以进行分析和检索。这些数据有着重大的意义，企业需要有一个专门的地方来分析它，以挖掘各种潜在的商机，这就是数据湖产生的原因。

企业数据主要分为 3 大类，具体如下：

- 主数据（master data），指的是详细描述企业内部主要实体的数据。通过观察主数据，可以了解企业涉及的业务。这些数据通常由不同部门管理和掌握。其他类别的数据，如后文所述，需要利用主数据来产生价值。
- 事务数据（transaction data），指的是各种应用程序（内部或外部）在处理企业内的各种业务流程时产生的数据。事务数据也包括人员相关的数据，虽然某些时候并不属于业务数据，但这部分数据也非常重要。分析这部分数据，可以帮助企业优化业务。这些数据也依赖于主数据，并经常引用主数据。
- 分析数据（analytic data），实际上指的是来源于前两类数据的数据。这部分数据是对企业中的各种实体（主数据）的深入分析，同时结合事务数据，为企业提供积极的建议，经过必要的调研之后，这些建议可以被企业采纳。

前面已经解释过这些类型的数据对企业的重要性，大多数企业都有管理这几类数据的机制，通常叫作企业数据管理（Enterprise Data Management，EDM）。下一节中将更详细地解释这方面的内容。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

图 1-2 显示了各种类型的企业数据以及它们是如何相互影响的。

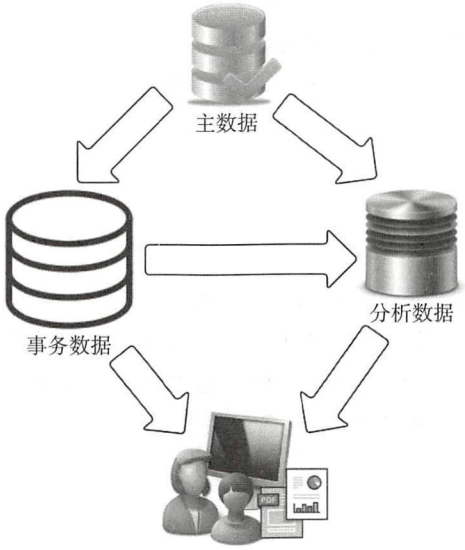


图 1-2 不同类型的企业数据

如图 1-2 所示，事务数据和分析数据都利用了主数据。分析数据也依赖于事务数据，并根据不同客户的需求进行有意义的分析。

1.3 企业数据管理

组织具有能够精确定义数据、轻松集成数据和同时为内部应用和外部通信检索数据的能力。

——维基百科

EDM 强调数据的精确性、粒度和含义，关注数据内容如何整合到业务应用程序中，以及数据如何从一个业务流程转移到另一个业务流程。

——维基百科

正如维基百科的定义中明确指出的那样，EDM 是定义了企业数据如何存储、存储的位置以及使用哪些技术来存储和检索这类数据的一整套策略。此类数据价值巨大，必须处于正确的控制下以保证安全，并且需要以明确的方式进行掌握和管理。EDM 还定义了数据如何与内部和外部的应用程序进行通信。此外，数据交换的相关政策和流程也必须明确。

从前文来看，对企业来说实现 EDM 非常容易，但实际上却是困难重重。一个企业有多个部门，每个部门都产生数据；基于这些部门的重要性，它们所产生的数据与整个组织密切相关。由于彼此处境不同，不同数据的所有者在 EDM 中有着不同的利益点，这会产生冲

仅供非商业用途或交流学习使用

突，从而给企业带来危害。这就需要在 EDM 中制定各种政策和程序来明确数据的所有权。

在本书的背景下，了解企业数据、企业数据管理以及与维护 EDM 相关的问题非常重要。这也是本书开篇就要介绍这些方面的原因。在下面的章节中，将介绍大数据的概念，以及如何将大数据整合到企业数据管理中，并扩展其功能。没有大数据技术的支持，这些扩展功能是很难实现的。

1.4 大数据相关概念

本节以维基百科对“大数据”一词的定义开始：

“大数据”是一个术语，用来描述这样一种数据集，它极其庞大、复杂，以至于传统的数据处理应用程序难以应付。术语“大数据”通常指的是应用预测分析，用户行为分析，或者某些其他从数据中提取有价值信息的高级数据分析方法，并不是专指某种特定规模的数据集。

——维基百科

下面我们来解释一下维基百科定义中给出的这两句话。早期，大数据本质上是指数据量很大且很复杂的数据。然而并没有任何明确的定义，指定何种规模的数据可以称为大数据。由于这种数据的数据量过于巨大，以至于传统的数据处理应用程序难以有效地使用它。在过去 10 年左右的时间中，这个领域出现了很多技术可以用来分析企业中的大数据。目前，“大数据”这个术语是指任何能理解和提取这些复杂企业数据，并对其进行有效利用的方法。

大数据和 4V

大多数时候，当读者接触到“大数据”这一术语时，都会遇到一个重要的概念——4V（此前是 3V，直到最近才引入了意义重大的第 4 个 V）。这 4V 分别指的是多样性（variety）、处理速度（velocity）、数据量（volume）和精准性（veracity），大数据对这 4 个维度的特征都有其独特的要求：

- 多样性：在大数据的背景下，多样性具有非常重要的地位。这里的多样性指的是丰富的数据类型和海量的数据源。随着技术的不断发展和应用程序（企业级和个人级）不断增加，数据多样性日益被重视。这种现象不仅不会减少，反过来，肯定还会在一段时间内持续增长。数据类型可以大致上分为结构化数据和非结构化数据。目前，大多数应用程序都是跟存储在关系型数据管理系统（RDBMS）中的结构化数据打交道。这种情况很普遍，但是现在，更多的需求转向了非结构化数据，如视频、图片、二进制文件等。
- 处理速度：在大数据的背景下，处理速度有两方面含义。首先是生成数据的速度，其次是需要快速实时地分析海量数据，从而推导出有意义的结论。正如谚语所说，

仅供非商业用途或交流学习使用

时间就是金钱。这个 V 的重要意义在于能够实时、快速地做出决策。一些企业，特别是零售业，非常倚重这方面的决策。如果顾客能适时看到适合自己的商品，那么他们很可能会购买。

- 数据量：在大数据的背景下，需要分析大量的数据，进而推导出有意义的结论。目前，并没有一个具体的数量来说明何等量级的数据可以被认为是大数据。但通常来说，这个数据量肯定显著高于传统应用能处理的数据量。一般来说，大数据对应的数据量相当大，而且确实让传统的应用程序（如 OLTP——联机事务处理）感到无能为力。对于很多业务来说，分析和利用社交数据不可或缺。这些社交应用（如 Facebook、Google +、LinkedIn 等）拥有十亿级的注册用户，每天会产生数十亿的数据（包括结构化及非结构化数据）。此外，应用自身的常规事务和其他分析（用户行为、位置信息等）也产生大量数据。另外，随着可穿戴设备和传感器的数量不断增长，这些设备又随时都在发送数据，数据量维度日益重要，而且这个趋势短时间内不会改变。

如前文所述，不久之前还只有 3 个 V。最近 IBM 提出了第 4 个 V，即精准性。由于数据呈指数级增长，并且数据来源的可靠性难以保障，因此这个 V 意义非凡。

当世界上有重大事件发生时，读者一定听说或阅读到过各种社交媒体上散布的假新闻。这个 V 指向一个非常重要的方面，那就是大数据中的精准性。随着数据大量涌现，尤其在社交网络中，该维度愈发重要，所以业界更倾向于 4V 而不是 3V。

- 精准性：在大数据的背景下，精准性指的是准确地分析数据，以获得有意义的结果。由于数据源众多，尤其某些用户输入的非结构化数据的质量不太可靠，处理某些渠道的数据时必须慎之又慎。如果有企业想要在业务上应用这些数据，必须更加严格地验证这些数据的准确性。

为了方便读者理解，图 1-3 清楚地显示了什么是 4V 以及各自的含义。

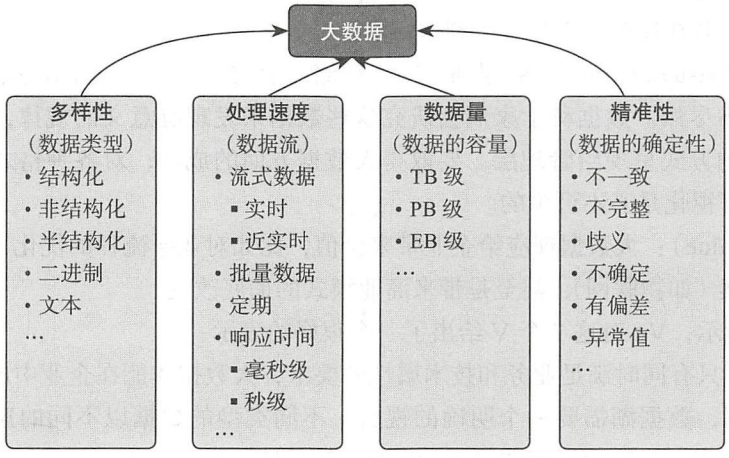


图 1-3 大数据的 4V

仅供非商业用途或交流学习使用

1.5 数据与企业的相关性

任何企业都不能否认数据的重要性。企业一直都在收集大量的历史数据，并将其保存在数据仓库中以便进行分析。企业可以通过分析历史数据来获得新的增长点，这也证明了数据的重要性。在过去的 10 年里，随着社交媒体的激增和海量应用（企业内部或外部云端）的涌现，企业收集的数据量呈指数级增长。随着数据量与日俱增，想要有效地利用这些海量的数据变得无比困难。企业试图利用这些收集到的数据来对现有业务转型或赋能，因此，数据的相关性是企业最关心的问题。

如果商业用户能获得大量的数据并使用正确的工具进行分析，那么他确实能够从中获得有用的价值。举例来说，如果能把来自各个应用的客户相关的数据汇聚到一起，并对其进行分析，则可以从这些数据中发现大量有价值的信息，比如哪些用户和企业网站有交互以及是怎样交互的。企业可以依据这些推导出的信息来改进现有的商业模式或者调整某些业务流程，以实现利润最大化。例如，可以基于集中的用户数据来思考新的商业模式，比如向用户提供积分。这些数据也可以用来向客户做个性化推荐。比如说，比起通用的推荐，基于用户行为向用户提供个性化的选项更容易被用户接受。当然，这些完全取决于具体业务，并不存在一种方法能适合所有的场景。不过，数据经过转换和清洗，再由各种数据可视化技术展示成丰富多样的图表，对商业用户来说更有用。

显然，企业数据与业务息息相关，下一节将详细介绍这些数据分布在企业的哪些地方。

Vit Soupal，德国电信（Deutsche Telekom AG）大数据部门负责人，在他的一篇博客中定义了 4V 作为大数据的技术属性，还另外引入了 3 个 V。本书并不打算完全涵盖这额外的 3 个 V，不过，想要在企业中成功应用数据湖（大数据），很难绕过它们。

另外 3 个 V（业务属性）如下：

- 视野（vision）：每一个拥有大数据（数据湖）的企业都应该有一个清晰的视野，并且改进业务流程来充分利用它。同时，企业的管理层应该充分理解数据湖给企业带来的变化，并在决策时充分考虑到这一点。
- 可视化（visualization）：数据湖预期将存储巨量数据。其中部分数据意义重大，另一部分则不尽然。数据科学家们会研究这些数据来发现有意义的规律，这些发现需要以有效的方式呈现给管理层。要取得大数据方面的成功，对各种格式的数据进行有意义的可视化是必不可少的。
- 价值（value）：大数据理应给企业带来价值，比如对业务流程的优化，或者创造新的解决方案（如物联网），甚至是带来商业模式的彻底改变。

如图 1-4 所示，Vit 对这 7 个 V 给出了一个很好的表示：

图 1-4 中，只有同时满足业务和技术属性的要求，大数据才能在企业中成功应用。

图 1-4 表明，数据湖需要一个明确的视野，不同类型的数据以不同的速度、不同的数据量流入这个湖中。进入数据湖的数据的质量（精准性）参差不齐。企业各个部门甚至高管

层都需要借助各种可视化方法来利用数据湖中的数据。这些可视化方法将为企业带来巨大的价值，它们能帮助企业做出各种有益的决策。技术属性（多样性、处理速度、数据量、精准性）固然不可或缺，但业务属性（视野、可视化、价值）也不容忽略，它们共同成就了数据湖。

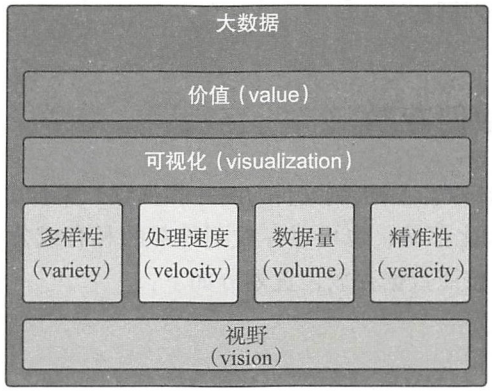


图 1-4 大数据的 7V

1.6 数据质量

毫无疑问，高质量的数据（清洁数据）是组织不可或缺的重要资产。反过来，如果基于低质量的数据来进行决策，反而会损害企业的利益和品牌价值，而品牌价值一旦流失就难以恢复了。一般来说，如果数据有不一致、重复、歧义和不完整的情形，数据的可用性就下降了。如果对这些数据进行分析时体验不佳，业务用户就不会再考虑使用该数据了。这时大家才意识到第 4 个 V（精准性）的重要性。数据质量是基于对数据的评估，来确定它是否与预想的应用场景相匹配。数据质量的评测基于多个维度的属性。下面是其中比较重要的部分属性：

- 正确性 / 准确性（correctness/accuracy）：表征了收集的数据所能描述客观世界的真实程度。
- 完整性（completeness）：通过统计获取的数据包含了多少预期 / 预定的属性。
- 一致性（consistency）：从多个系统中获取数据，并汇聚起来在一起对比，以此来衡量数据的一致性。
- 时效性（timeliness）：表征了在指定的时间内及适当的场景中，向正确的人提供高质量数据的能力。
- 元数据（metadata）：根据获取的数据中的附加数据来衡量。正如这个术语本身表明的那样，元数据是描述数据的数据，它可以用来定义数据本身，可以使数据更便于使用。

- 数据世系（data lineage）：对数据的整个生命周期进行跟踪，可能会给组织来带巨大的收益。数据的这种可追溯性能为组织提供十分有价值的商业见解。

除了前文所述的，其实还有其他的维度可以用来表征数据的质量。虽然这里只对上述部分做了详细介绍，此时读者的脑海中应该已经有了一个清晰的概念，在阅读本书后续章节时，读者会有更清晰的理解。

1.7 企业中数据如何存放

企业数据以原始数据格式、二进制文件（图片和视频）等形式存储，存放的位置可以是各种内部应用程序的持久化存储，也可以存放到外部的私有或公有的云平台中。首先，需要对数据类型进行划分。按照数据存放位置划分如下：

- 内联网（企业内部）
- 互联网（企业外部）

另一种方式是按照数据存储的格式划分，具体如下：

- 持久化存储引擎（RDBMS 或者 NoSQL）
- 传统的数据仓库（基于 RDBMS、NoSQL 等）
- 文件存储

下面进一步详细介绍这些数据类型。

1.7.1 内联网（企业内部）

企业数据存放在自己的私有网络，即内联网中。

企业内部的各种应用程序也运行在私有网络中，并且只有指定的人员才能访问。因此，从这些应用程序获取的数据能以安全和私密的方式存储于企业内部。

这些应用产生的数据可能是员工数据，也可能是日常使用中产生的事务数据。

企业用来构建内联网的技术包括局域网（LAN）和广域网（WAN）。此外，企业内部还有多个应用平台可以使用，数据可以以结构化方式存储在不同的存储引擎中，例如，传统的 RDBMS 和 NoSQL 数据库。除了这些存储引擎，还有以各种文件形式存放的非结构化数据。另外，大数据企业都有传统的数据仓库，用来存放清洗后的数据以备分析之用。

1.7.2 互联网（企业外部）

大约从近 10 年开始，大多数企业都拥有自己的数据中心，几乎所有的企业数据都存放于此。伴随云计算的发展，企业开始把一些数据存放到私有数据中心以外的云平台上，加上健全的安全控制，未经授权的人根本接触不到这些数据。采用云服务的方式能极大地降低企业的运营成本，这是云服务最大的优势。本节将进一步介绍这个领域，包括更细粒度

的类别划分。

1. 云托管业务应用程序

云计算供应商提供了各种服务，如 SaaS、PaaS、IaaS 等，企业可以把业务应用程序托管到云中，同样也能满足企业管理的需求。因此，许多企业选择把内部开发的应用程序托管到这些云计算提供商。员工使用这些部署在云中的应用程序，进行日常的一些操作，这和操作部署在企业自有的数据中心中的应用没有区别。

2. 第三方云计算方案

如今这么多公司把自身的应用 / 服务托管到云端，这些应用 / 服务用起来就跟部署在本地一样，但是企业不必考虑维护和管理应用 / 服务所需的基础架构。这些仅仅依赖云端环境的产品，最吸引企业的一点是它们的收费方式。

基于这个优点，企业倾向于选择这些云产品，同时也由于云产品的先天特点，企业现在也将数据（业务相关）存储在其他的云设施中，即使云提供商完全掌控了这些数据。

Google 的 BigQuery 就是这样一款云服务软件，用户可以把数据导出到云服务中，然后运行这个软件来进行各种分析工作。这类产品有一个优点，就是我们可以根据分析的结果来决定是否保留这些数据。云服务具有天然的可伸缩性（硬件层面随意扩展和收缩的能力），如果数据分析很复杂，可以临时申请性能强劲的服务器，分析结束之后又可以把服务器的配置恢复到此前的水平。

基于这个特性，Google BigQuery 将自己定位为“企业级云数据仓库”，这也确实是实至名归。它给企业带来了快速分析大规模数据的能力，同时保证了安全性、可靠性和可用性。它还集成了其他类似的云服务软件，以满足各种其他需求。

Google 的 BigQuery 仅仅是一个例子，还有许多其他的云服务软件，具有不同程度的类似特征。现在的企业都在追求业务开展速度，既不想在工具性软件上投入时间进行研究，也不愿意投入经费来管理基础设施；这些云服务的解决方案满足了企业的全部需求，而经济和精力上的成本也相对低廉。

当前阶段，类似的解决方案仍在不断地涌现，本书不再一一罗列。所以仅以 BigQuery 为例来介绍这一类方案。

与 SaaS 的方式类似，云端还有很多业务应用程序也可以作为服务对外提供。Salesforce 就是这样的一个例子。本质上，Salesforce 是一个客户关系管理（CRM）的解决方案，但是它还集成了很多其他特性。笔者在这里并不是想帮助推销 Salesforce，但是希望介绍一些云端业务应用的重要特性。Salesforce 将所有的客户信息汇总到一起，帮助企业从销售、业务分析和客户服务等几个维度来构建以用户为中心的商业模式。

因为部署在云端，Salesforce 也带来了很多云端 SaaS 的特性。

随着云服务对企业的影响越来越大，现在大量企业把数据存储在互联网（云）上，同时，这些云服务要明确保护客户隐私和其他共同权益，以捍卫企业的商业目标。

3. 社交数据（结构化与非结构化）

如今，企业的社交关系变得至关重要，即使企业的数据并没有存储在社交网站上，它仍然包含了丰富的信息，这些信息来自企业业务和其服务的真实客户。

用户在社交网站上留下各种评价和建议，企业可以利用这些数据重塑自己的业务和与用户互动的方式。

如果企业忽视此类信息，很可能会损害企业的声誉和品牌。现在企业非常重视社交网站，因为这些网站上即使没有企业的数据，也有客户的评价信息，在某种程度上，这些信息代表了客户对品牌的认知。

于是，我们倾向于将这部分数据归类为非企业用户提供的企业数据。在分析这类数据时，需要特别关注大数据的第 4 个 V，即精准性。因为有些人会利用这个渠道，在与企业进行业务往来时，获得一些不正当的优势。

另一个对企业数据进行分类的方式是按数据最终的存储方式。下一节将更详细地介绍这种分类。

1.7.3 数据持久化存储（RDBMS 或者 NoSQL）

无论是在企业内部（企业基础架构）还是在云服务中，数据通常都以结构化数据的方式存储在所谓的传统 RDBMS 或者新一代 NoSQL 持久化存储中。数据通过业务应用程序写入这些存储系统，绝大部分数据在本质上是不相关的，企业可以很容易理解这些数据。数据存储在传统的 RDBMS 类存储系统中时，会遇到的一个常见的问题，即数据量超出了可接受的范围。在这种情况下，对数据进行分析将耗费大量的时间和精力。因此，企业被迫将数据分拆为生产数据（数据可以被业务应用查询和使用）和非生产数据（从生产系统转移到另外的存储系统中的离线数据）。

由于这种拆分，数据分析通常跨越若干年，这种分析并不能覆盖到企业的全部业务因素。举例来说，如果生产系统中有 5 年的销售数据，而非生产系统存储了 15 年的销售数据，那么用户在进行销售数据分析时，只能查看过去 5 年的数据。可能存在周期刚好为 5 年的变化趋势，只有对 20 年的销售数据进行分析才能发现。大多数时候，由于 RDBMS 的局限，存储和分析海量数据是不可能的。即使勉力为之，这种分析也十分耗时，也不能给分析师提供足够的灵活性。这样一来，分析师只能进行有限的分析，如果企业正在通过这些数据来调整业务流程，那么可能会给企业带来大麻烦。

新一代 NoSQL（该领域中不同的数据库具有不同的功能）让数据分析更灵活，同时存储数据的量也大大增加。这些存储满足了分析师们在性能以及其他某些方面的需求，但是在一些方面仍然存在不足。

即使数据存储在一个业务应用中，各种业务应用的数据并没有汇聚成一个单一的视图，而建设一个合适的数据湖能帮企业解决这种问题。

1.7.4 传统的数据仓库

正如前一节中介绍的那样，由于生产系统的业务应用获取的数据量巨大，几乎所有的时间里，生产系统的数据都与非生产系统分离。非生产系统的数据通常存储在不同的区域中，并保存在不同的数据存储系统（通常是RDBMS或者NoSQL）中，这种存储系统就是所谓的数据仓库。通常来说，根据数据分析师的需求，数据会被相应地清洗和删除。删除部分数据会限制分析师能执行的分析的类别。大多数情况下，都会有隐藏的有价值的数据尚未进入数据仓库，这就会导致更多的分析，而企业可以基于这些分析来调整业务；然而，此类数据清理和删除可能会扼杀潜在的创新。因此这方面需要改进。而本书介绍的数据湖，使得分析师可以引入生产系统中的任意业务应用数据，并根据具体情况进行各种分析。

目前，构建这类数据仓库的方案是通过ETL（Extract,Transform,Load，即抽取、转换、加载），从生产系统的数据库提取数据并保存到数据仓库中。ETL流程会根据分析师的需求来清洗数据，而分析师会在数据仓库中进行各种分析。

1.7.5 文件存储

业务应用程序在不断地变化，新的应用允许终端用户以各种格式获取数据，而不仅仅限于从键盘输入的格式化数据。

终端用户另一种输入数据的渠道是提交不同格式的文档。其中一些众所周知的格式如下：

- 文档（PDF、DOC、XLS等）
- 二进制
  - 图片（JPG、PNG等）
  - 音频（MP3、RAM、AC3）
  - 视频（MP4、MPEG、MKV）

正如前文中介绍的那样，处理结构化数据本身就是一个大问题，现在又引入了非结构化数据的分析。但是，分析非结构化数据与分析结构化数据同等重要。通过构建数据湖，我们可以围绕数据湖引入新的技术，利用大数据领域最新和最好的技术，可以从非结构数据中挖掘到巨大价值。

除了各种文件格式的数据，有很多应用允许终端用户以语句的形式获取大量数据，这部分数据也需要分析。手动处理这些来自终端用户的评论数据是一项艰难的任务，在现代化的今天，需要的是自动化地解析这些评论，并了解用户的观点。同样，有很多这样的技术可以理解这些数据（如自由文本流），让企业能够正确地处理它。

举例来说，假如我们有一个获取企业建议的系统，每天能获取近1000条建议，由于该业务的天然属性，很难对这些建议进行过滤分析。这种场景中，我们可以利用一些技术来对这些评论进行情感分析，然后根据这些分析工具给出的评分进行初步的过滤，再将筛选后的建议交给能够理解和利用这部分数据的人来处理。

1.8 企业现状

正如前面几节简要介绍的那样，企业数据当前的状态可以总结为以下几点：

1. 传统 DW（Data Warehouse）/BI（Business Intelligence）

- 使用 ETL 清洗来自生产业务应用的数据。
- 早于一定时间的数据会转移到另外的存储系统（如磁带）中，但是很难对这部分数据进行检索。
- 一些显著的缺陷：
  - 数据仓库中保存的是清洗后的生产数据的子集；在数据仓库中添加任何数据元素都需要付出代价。
  - 数据仓库中保存的是全量数据的一个子集，其余数据会转存到另外的持久存储系统中去。
- 通常分析速度较慢，即使对查询做了一定程度的优化。

2. 大数据孤岛

- 一些部门在建设大数据时采取了正确的措施。但是部门间通常不会互相协作，这些大数据平台就成了孤岛，并没有真正给企业带来价值。
- 一些显著的缺陷：
  - 大数据平台的孤立性再次限制了分析师，使得分析师们无法跨部门整合和查询数据。
  - 需要大量资金来构建、维护、管理这些数据孤岛，时间一长就难以为继。

3. 大量非连接应用

- 在企业内部和云服务中部署大量应用程序。
- 除了产生结构化数据，应用程序也产生非结构化数据。
- 一些显著的缺陷：
  - 互相没有通信。
  - 即使有通信，数据科学家们既不能有效地利用这些数据，也不能对企业进行有益的改进。
  - 各个业务应用在多个方面重复使用各种技术。

这里并不是说投资构建数据湖是解决上述所有缺陷的“银弹”。但是这个方向一定是正确的，企业至少应该花时间去思考是否确实需要数据湖，如果答案是肯定的，就不要过多考虑，直接着手去构建。

构建数据湖是企业的一项战略级决策，在构建时需要所有相关人员的认可与支持，而且需要高管层投入精力关注这个项目。基于数据湖，企业一定能找到改进业务流程的方法。它能让高管人员更深入地了解业务，同时也能提升决策的成功率。

1.9 企业数字化转型

数字化转型指的是应用数字化技术，从根本上影响商业和社会的方方面面。

——infoworld.com

数字化转型（DX）是一个流行的行业术语，每个企业都毫不犹豫地跟随了这个趋势。这个术语指的是企业向以信息技术为核心转型。技术领域的投入是这项转型的一部分，但数据才是转型的关键。

近年来，企业越来越多地认识到数据和数据分析的重要性，这无疑让企业有了新思维。这种转型就是一种以数据为中心的实践。

作为商业转型的一部分，企业绝对应该把数据湖列入重点投入项目列表，所有部门都应该欣然将数据归集到数据湖来完成数据共享，而不能对它存有偏见。

企业转型之路

埃森哲互动曾经委托 Forrester 咨询公司进行过一项研究，该研究表明数字化转型的关键驱动力是提升盈利能力、客户满意度和产品迭代速度。

实际上，很多企业已经走上了数字化的转型之路。数字化转型不再是一句空话，很多企业正竭力转向把技术作为驱动力之一，而另一个驱动力，如读者猜测的那样，就是数据。

转型中的企业有明确的目标。当然，这些目标与它们所处的具体行业有关。但是有些目标是公共的，具体如下：

- 从根本上改善客户体验。
- 削减成本。
- 增加收入。
- 差异化竞争。
- 调整业务流程，反过来改善商业模式。

转型目标举例

关于企业在该领域希望达到的目标，有一系列清晰的例子，部分如下：

- 能够对客户进行细分并提供个性化的产品。在恰当的时间向合适的客户推送广告。
- 引进更多的技术，减少体力劳动，企业实现全方位数字化。
- 结合社交信息和企业自身数据，为决策提供支持。
- 以更为量化的方式预测未来，并采取必要措施，提前做好预案。
- 以技术为重要载体实现业务全球化。

下一节中将详细介绍企业在数字化转型中希望实现的一个用例，其中数据是主要的贡献因素。理解该用例很重要，因为我们会在本书中尝试实现该用例。

1.10 数据湖用例启示

数据在企业中的重要性已经显露无遗。当今企业所面临的问题是如何从这些数据中挖掘有用信息来优化自身的业务。

即便数据能够以某种方式汇聚到一起，也很难在合适的时间内处理数量如此巨大的数据。这时数据湖进入了人们的视野。下一章中将全面而详尽地介绍数据湖到底是什么。在那之前，我们会详细介绍本书中将要实现的用例，这个用例是以数据湖为中心的。

基于现代技术构建数据湖有很多优势，其中一些如下：

- 通过各种数据分析，企业能找到业务中重要的人员和流程，也能深入了解各类客户。
- 企业能在适度的时间范围内完成这些分析，而不用等待几周甚至几个月。
- 业务用户能快速分析数据，进而快速调整业务流程。

本书中涵盖的这个用例叫作单一客户视图（Single Customer View，SCV）。单一客户视图是业内众所周知的术语，它有很多种定义，下面是其中之一：

> 单一客户视图是组织或企业的客户数据的一种聚合的、一致的及全面的表示。
>
> ——维基百科

企业将客户数据保存在不同的业务应用中，这会导致不同程度的数据孤岛现象。该用例旨在将这些不同来源的数据整合到一起，提供给分析人员查看，从而创建一个包含所有数据的客户视图。通过这个视图，企业能对客户进行细分，进而向客户进行精准广告投放。

这个用例对企业的意义可以归纳为如下几点：

- 客户细分。
- 信息整合。
- 增进客户关系，避免客户流失。
- 更深入的分析等。

从概念上讲，图 1-5 中总结了本书计划实现的用例。结构化、半结构化和非结构化数据都被灌入数据湖。单一客户视图从数据湖中以整体性方式被导出。该用例会引入各类数据，也会详细描述各种类型的数据样本。之所以这样做，是因为这样可以充分利用数据湖，同时也更为接近真实情况。

图 1-5 中显示了数据湖从各种数据源（多样性），以不同的速度获取不同数量级的数据。更多地，这是一个概念性的抽象视图，在读完本书的全部内容后，我们将会确切知道它具体指代的是什么。

此刻，笔者真的很兴奋，因为马上要开始激动人心的阅读之旅了，希望读者也能兴奋起来！

非卖品，仅供非商业用途或交流学习使用

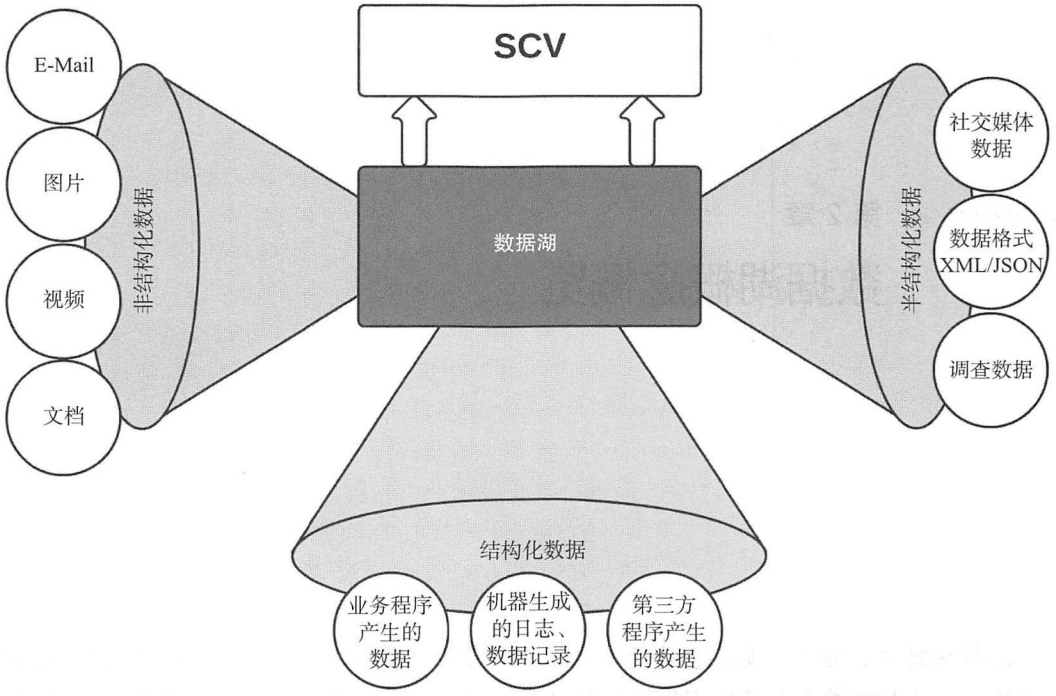


图 1-5　SCV 数据湖用例的概念视图

1.11　总结

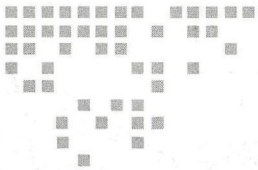
本章带领读者深入了解了一些常用的行业术语。在开头的部分，读者理解了数据、企业数据以及最重要的大数据。然后介绍了数据与企业的相关性，包括数据质量的各个属性。接着，本章详细介绍了不同的数据类型，以及企业中数据存储的位置。

接下来的部分中探讨了数字化转型，并详细介绍了本书将要实现的用例。

阅读完本章后，读者能清晰地掌握许多术语，也会认识到数据和数据湖对企业的重要性。此外，读者能清晰地理解本书给出的用例。

在下一章中，我们将深入探究数据湖，以及构建数据湖时会用到的模式。

非卖品，仅供非商业用途或交流学习使用



Chapter 2 第 2 章

数据湖概念概览

数据湖概念的诞生，源自企业面临的一些挑战，如数据应该以何种方式处理和存储。最开始，企业对种类庞杂的应用程序的管理都经历了一个比较自然的演化周期。最开始的时候，每个应用程序会产生、存储大量数据，而这些数据并不能被其他应用程序使用，这种状况导致数据孤岛的产生。随后数据集市应运而生，应用程序产生的数据存储在一个集中式的数据仓库中，可根据需要导出相关数据传输给企业内需要该数据的部门或个人。然而数据集市只解决了部分问题。剩余问题，包括数据管理、数据所有权与访问控制等都亟须解决，因为企业寻求获得更高的使用有效数据的能力。为了解决前面提及的各种问题，企业有很强烈的诉求搭建自己的数据湖，数据湖不但能存储传统类型数据，也能存储任意其他类型数据，并且能在它们之上做进一步的处理与分析，产生最终输出供各类程序消费。在本章中，将介绍数据湖的一些主要方面，帮助读者理解为什么它对企业非常重要。

2.1 什么是数据湖

如果需要给数据湖下一个定义，可以定义为这样：数据湖是一个存储企业的各种各样原始数据的大型仓库，其中的数据可供存取、处理、分析及传输。

数据湖从企业的多个数据源获取原始数据，并且针对不同的目的，同一份原始数据还可能有多种满足特定内部模型格式的数据副本。因此，数据湖中被处理的数据可能是任意类型的信息，从结构化数据到完全非结构化数据。企业对数据湖寄予厚望，希望它能帮助用户快速获取有用信息，并能将这些信息用于数据分析和

非卖品，仅供非商业用途或交流学习使用

机器学习算法，以获得与企业运行相关的洞察力。

数据湖与企业的关系

数据湖能给企业带来多种能力，例如，能实现数据的集中式管理，在此之上，企业能挖掘出很多之前所不具备的能力。另外，数据湖结合先进的数据科学与机器学习技术，能帮助企业构建更多优化后的运营模型，也能为企业提供其他能力，如预测分析、推荐模型等，这些模型能刺激企业能力的后续增长。

企业数据中隐藏着多种能力，然而，在重要数据能够被具备商业数据洞察力的人使用之前，人们无法利用它们来改善企业的商业表现。

2.2　数据湖如何帮助企业

长期以来，企业一直试图找到一个统一的模型来表示企业中所有实体。这个任务有极大的挑战性，原因有很多，下面列举了其中的一部分：

- 一个实体在企业中可能有多种表示形式，因此可能不存在某个完备的模型来统一表示实体。
- 不同的企业应用程序可能会基于特定的商业目标来处理实体，这意味着处理实体时会采用或排斥某些企业流程。
- 不同应用程序可能会对每个实体采用不同的访问模式及存储结构。

这些问题已困扰企业多年，并阻碍了业务处理、服务定义及术语命名等事务的标准化。

从数据湖的角度来看，我们正在以另外一种方式来看待这个问题。使用数据湖，隐式实现了一个较好的统一数据模型，而不用担心对业务程序产生实质性影响。这些业务程序则是解决具体业务问题的“专家”。数据湖基于从实体所有者相关的所有系统中捕获的全量数据来尽可能“丰满”地表示实体。

因为在实体表示方面更优且更完备，数据湖确实给企业数据处理与管理带来了巨大的帮助，使得企业具备更多关于企业增长方面的洞察力，帮助企业达成其商业目标。值得一提的是，Martin Fowler 写过一篇很有意思的文章，在这篇文章中，他对企业数据湖的一些关键方面做了简明扼要的阐述，可参考下面这个链接：https://martinfowler.com/bliki/DataLake.html。

数据湖的优点

企业会在其多个业务系统中产生海量数据，随着企业体量增大，企业也需要更智能地处理这些横跨多个系统的数据。

一种最基本的策略是采用一个单独的领域模型，它能精准地描述数据并能代表对总体

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

业务最有价值的那部分数据。这些数据指的是前面提到的企业数据。

对企业数据进行了良好定义的企业当然也有一些管理数据的方法，因此企业数据定义的更改能保持一致性，企业内部也很清楚系统是如何共享这些信息的。

在这种案例中，系统被分为数据拥有者（data owner）及数据消费者（data consumer）。对于企业数据来说，需要有对应的拥有者，拥有者定义了数据如何被其他消费系统获取，消费系统扮演着消费者的角色。

一旦企业有了对数据和系统的明晰定义，就可以通过该机制利用大量的企业信息。该机制的一种常见实现策略是通过构建企业级数据湖来提供统一的企业数据模型，在该机制中，数据湖负责捕获数据、处理数据、分析数据，以及为消费者系统提供数据服务。数据湖能从以下方面帮助到企业：

- 实现数据治理（data governance）与数据世系。
- 通过应用机器学习与人工智能技术实现商业智能。
- 预测分析，如领域特定的推荐引擎。
- 信息追踪与一致性保障。
- 根据对历史的分析生成新的数据维度。
- 有一个集中式的能存储所有企业数据的数据中心，有利于实现一个针对数据传输优化的数据服务。
- 帮助组织或企业做出更多灵活的关于企业增长的决策。

在本节中，我们讨论数据湖应该具备哪些能力。本章后续部分将会讨论和评述数据湖是如何工作的，以及应该如何去理解其工作机制。

2.3 数据湖是如何工作的

为了准确理解数据湖能给企业带来哪些好处，理解数据湖的工作机制以及构建功能齐全的数据湖需要哪些组件就显得尤为重要了。在一头扎进数据湖架构细节之前，不妨先来了解数据湖背景中的数据生命周期。

在一个较高的层面来看，数据湖中数据生命周期如图 2-1 所示。

上述生命周期也可称为数据在数据湖中的多个不同阶段。每个阶段所需的数据和分析方法也有所不同。数据处理与分析既可按批量（batch）方式处理，也可以按近实时（near-real-time）方式处理。数据湖的实现需要同时支持这两种处理方式，因为不同的处理方式服务于不同的场景。处理方式（批处理或近实时处理）的选择也依赖数据处理或分析任务的计算量，因为很多复杂计算不可能在近实时处理模式中完成，而在一些案例中，则不能接受较长的处理周期。

同样，存储系统的选择还依赖于数据访问的要求。例如，如果希望存储数据时便于通过 SQL 查询访问数据，则选择的存储系统必须支持 SQL 接口。如果数据访问要求提供数据

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

视图，则涉及将数据存储为对应的形式，即数据可以作为视图对外提供，并提供便捷的可管理性和可访问性。最近出现的一个日渐重要的趋势是通过服务（service）来提供数据，它涉及在轻量级服务层上对外公开数据。每个对外公开的服务必须准确地描述服务功能并对外提供数据。此模式还支持基于服务的数据集成，这样其他系统可以消费数据服务提供的数据。

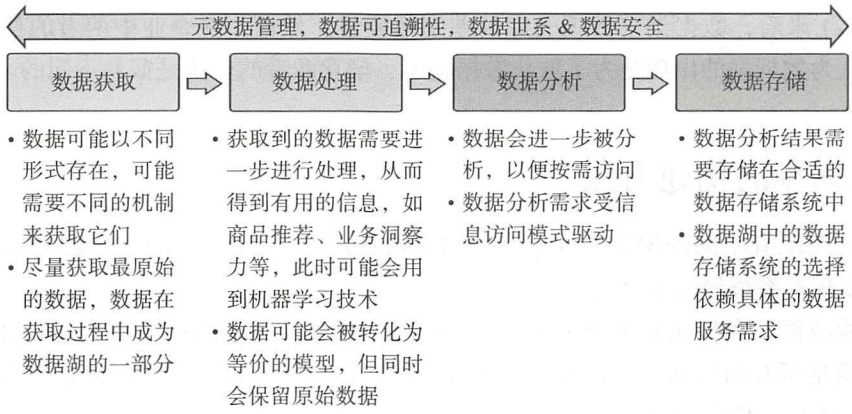


图 2-1　数据湖的生命周期

当数据从采集点流入数据湖时，它的元数据被捕获，并根据其生命周期中的数据敏感度从数据可追溯性、数据世系和数据安全等方面进行管理。

数据世系被定义为数据的生命周期，包括数据的起源以及数据是如何随时间移动的。它描述了数据在各种处理过程中发生了哪些变化，有助于提供数据分析流水线的可见性，并简化了错误溯源。

可追溯性是通过标识记录来验证数据项的历史、位置或应用的能力。

——维基百科

2.4　数据湖与数据仓库的区别

很多时候，数据湖被认为与数据仓库是等同的。实际上数据湖与数据仓库代表着企业想达成的不同目标。表 2-1 中显示了两者的关键区别。

表 2-1　数据湖与数据仓库的关键区别

数　据　湖	数 据 仓 库
能处理所有类型的数据，如结构化数据、非结构化数据、半结构化数据等，数据的类型依赖于数据源系统的原始数据格式	只能对结构化数据进行处理，而且这些数据必须与数据仓库事先定义的模型吻合

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

（续）

数 据 湖	数据仓库
拥有足够强的计算能力用于处理和分析所有类型的数据，分析后的数据会被存储起来供用户使用	处理结构化数据，将它们或者转化为多维数据，或者转换为报表，以满足后续的高级报表及数据分析需求
数据湖通常包含更多的相关信息，这些信息被访问的概率很高，并且能够为企业挖掘新的运营需求	数据仓库通常用于存储和维护长期数据，因此数据可以按需访问

从表 2-1 来看，数据湖与数据仓库的差别很明显。然而，在企业中两者的作用是互补的，不应认为数据湖的出现是为了取代数据仓库，毕竟两者的作用是截然不同的。

2.5 数据湖的构建方法

不同的组织有不同的偏好，因此它们构建数据湖的方式也不一样。构建方法与业务、处理流程及现存系统等因素有关。

简单的数据湖实现几乎等价于定义一个中心数据源，所有的系统都可以使用这个中心数据源来满足所有的数据需求。虽然这种方法可能很简单，也很划算，但它可能不是一个非常实用的方法，原因如下：

- 只有当这些组织重新开始构建其信息系统时，这种方法才可行。
- 这种方法解决不了与现存系统相关的问题。
- 即使组织决定用这种方法构建数据湖，也缺乏明确的责任和关注点隔离（responsibility and separation of concerns）。
- 这样的系统通常尝试一次性完成所有的工作，但是最终会随着数据事务、分析和处理需求的增加而分崩离析。

更好的构建数据湖的策略是将企业及其信息系统作为一个整体来看待，对数据拥有关系进行分类，定义统一的企业模型。这种方法虽然可能存在流程相关的挑战，并且可能需要花费更多的精力来对系统元素进行定义，但是它仍然能够提供所需的灵活性、控制和清晰的数据定义以及企业中不同系统实体之间的关注点隔离。这样的数据湖也可以有独立的机制来捕获、处理、分析数据，并为消费者应用程序提供数据服务。

2.6 Lambda 架构驱动的数据湖

正如我们在前面讨论的那样，有多种处理数据的方式，它们可以被大致地分为批处理和实时数据处理。虽然可能有这样的场景，只用其中一种处理方式就提供了所需的结果，但也有可能同时需要来自批处理和实时数据处理组件处理后产生的数据。这使我们遇到了将批处理结果与实时处理结果合并的问题。这个问题可由 Lambda 架构模式解决，细节将会在下一章中进行讨论。这里讨论的是最原始的由 Lambda 架构驱动的数据湖理念。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

Lambda 架构作为一种模式，提供了在大型数据集上执行高度可伸缩和高性能分布式计算的方法，并且最终为批处理和近实时处理提供了一致的数据。Lambda 架构定义了能应对企业中各种数据负载的可水平扩展架构的实现方法与手段，并且具有较低的延迟预期。图 2-2 显示了数据湖中的功能模块。

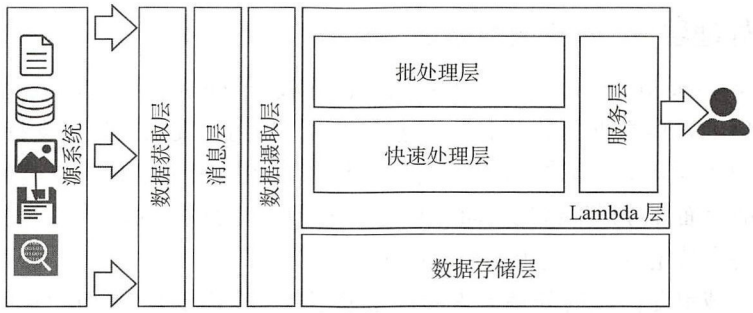


图 2-2　数据湖中的功能模块

Lambda 架构模式的实现方式是将整个架构划分为多个功能模块 / 层（layer）。每一层都将在本章的后续部分中进行简略介绍。

2.6.1　数据摄取层——摄取数据用于处理和存储

快速的数据摄取层（data ingestion layer）是 Lambda 架构模式中的关键层之一。这一层需要控制将数据快速传递到 Lambda 架构的工作模型中。该层的关键功能如下所列：

- 该层必须具有高度可扩展性，满足各种需求，能够根据不同的负载情况进行伸缩。
- 该层必须具有容错（fault tolerant）能力，提供系统可靠性和故障转移（fail-over）能力。
- 该层必须支持多线程及多事件处理。
- 该层必须能够快速地将所摄取数据的结构转换为目标数据格式，这是 Lambda 架构处理层所要求的。
- 该层必须确保所交付的所有数据都以最纯粹的形式供下一步处理。

2.6.2　批处理层——批量处理已提取数据

批处理层（batch layer）是 Lambda 架构中对已提取数据进行批量处理的层，以确保系统资源的最佳利用，同时也可将长时间运行的操作应用于数据，以确保输出数据的高质量。输出数据也称为模型数据（modeled data）。将原始数据转换为模型数据是批处理层的主要职责，其中，模型数据中蕴含了 Lambda 架构中服务层（serving layer）向外提供数据的数据模型。该层主要职责如下所列：

- 该层必须能在已摄取的原始数据之上执行数据清理、数据处理、数据建模算法。
- 该层必须提供重新执行（replay/rerun）某些操作的机制，以实现故障恢复。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

- 该层必须支持在已摄取的原始数据之上执行机器学习算法或数据科学处理，以产生高质量的模型数据。
- 该层可能需要执行一些其他操作，以期通过移除重复数据、检测错误数据和提供数据世系视图来提高模型数据的整体质量。

2.6.3 快速处理层——近实时数据处理

快速处理层（speed layer）将对从数据摄取层接收的数据执行近实时处理。由于处理预期接近实时，因此这些数据的处理需要快速、高效，为高并发场景提供支持和相应的精心设计，并且最终产生满足一致性要求的输出结果。很多因素可以影响快速处理层的特性，这些将在本书的后面部分详细讨论。简单来说，该层应包含以下功能：

- 必须支持在特定数据流之上的快速操作。
- 必须能生成满足近实时处理需求的数据模型。所有需要长时间运行的处理必须被委托给批处理模式。
- 必须有快速访问能力和存储层的支持，这样就不会因为处理能力而导致事件的堆积。
- 必须与数据摄取层的批处理过程分离。
- 必须产生一个输出模型，该模型（从某种程度上来说）可以与批处理产生的数据集合并，进一步提供增强型的企业数据。

2.6.4 数据存储层——存储所有数据

在 Lambda 架构模式中，数据存储层（data storage layer）非常引人注目，因为该层定义了整个解决方案对传入事件 / 数据流的反应。由架构常识可知，一个系统的速度最多与处理链中最慢的子系统一样快，因此，如果存储层不够快，由近实时处理层执行的操作将会变得很慢，从而阻碍了该架构达到近实时的效果。

在 Lambda 的总体架构中，针对已摄取的数据有两种主动操作：批处理和近实时处理。批处理和近实时处理的数据需求差别很大。例如，在大多数情况下，批处理模式需要执行串行读和串行写操作，此时使用 Hadoop 存储层就足够了，但是如果我们考虑近实时处理，需要快速查找和快速写入，那么 Hadoop 存储层可能是不合适的（见表 2-2）。为了支持近实时处理，需要数据层支持某些类型的索引数据存储。

表 2-2 Hadoop 存储层对批处理和近实时处理模式的适用情况

批 处 理	近实时处理
串行写及串行读操作	快速查找，快速写入
Hadoop 存储层能胜任	Hadoop 存储层不能胜任

Lambda 架构的典型功能如下所列：

- 同时支持串行读写及随机读写。
- 针对用户的使用情况，提供合适的层次性的解决方案。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

- 支持以批量模式或近实时模式处理海量数据。
- 以灵活、可扩展的方式支持多种数据结构的存储。

2.6.5　服务层——数据交付与导出

Lambda 架构也强调了为消费者程序提供数据传输服务的重要性。众所周知，数据可以以多种方式在系统间传递。其中最重要的一种方式是通过服务（service）传递。在数据湖背景中，这些服务被称为数据服务（data service），因为它们的主要功能是传输数据。

另外一种传输数据的方式是数据导出（export）。数据最终可导出为多种格式，如消息、文件、数据备份等，导出的数据供其他系统消费。

数据传输 / 服务主要关注的是如何将数据转换为预期的格式。这种格式可以强制约定为数据契约（data contract），数据服务在对外提供服务时遵循该约定。然而，在执行数据传输操作时，合并批量处理及近实时处理产生的数据非常重要，因为这两类数据中都可能包含与组织机构相关的关键信息。数据服务层必须保证数据与数据契约（与消费者程序约定）的一致性。

从较高的层次来看，数据服务层应满足下列特性：

- 支持多种机制为消费者程序提供数据服务。
- 每种支持数据服务的机制，必须与消费者程序的数据契约兼容。
- 支持批量处理及近实时处理数据视图的合并。
- 为消费者程序提供可扩展、快速响应的数据服务。

因为数据服务层的核心职责是向数据湖以外的消费者提供数据服务，出于增强数据表现的考虑，该层可能会选择性地进行数据合并。

以上是 Lambda 架构的主要功能模块，其他向 Lambda 架构灌入数据供其处理的功能模块，如数据获取层（data acquisition layer）、消息层（messaging layer）及数据摄取层（data ingestion layer）等将在本章后续小节中讨论。

2.6.6　数据获取层——从源系统获取数据

企业中数据格式多种多样，可大致分为结构化数据、半结构化数据和非结构化数据。

结构化数据的常见例子包括关系数据库、XML/JSON、系统间传递的消息等。企业也非常青睐半结构化数据，尤其是 E-Mail、聊天记录、文档等。非结构化数据的典型例子包括图片、视频、原始文本、音频文件等。

对于这些类型的数据，部分数据可能无法对其定义模式（schema）。需要将数据转换为有意义的信息时，模式是非常重要的。为结构化数据定义模式的方法非常直接，但是无法为半结构化数据或非结构化数据定义模式。

数据获取层的一个关键作用是将数据转换为在数据湖中可进行后续处理的消息。因此数据获取层必须非常灵活，能适应多种数据模式。同时，它也必须支持快速的连接机制，

仅供非商业用途或交流学习使用

无缝地推送所有转换过的数据消息到数据湖中去。

数据获取层在数据获取端由多路连接（multi-connector）组件构成，然后将数据推送到特定的目的地。在数据湖的例子中，目的地指的是消息层，如图 2-3 所示。

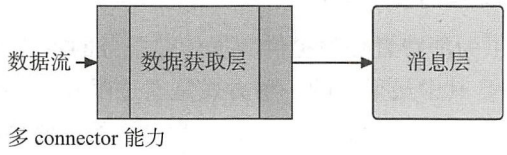


图 2-3 数据获取组件

很多技术框架可以用于构建能支持多种源系统的低延迟的数据获取层。对于每种源系统类型，数据获取层的连接都需要根据所依赖的底层框架进行特殊配置。数据获取层会对已获取的数据做少量转换，其目的是最小化传输延迟。这里的数据转换指的是将已获取的数据转换为消息或事件，它们可以发送给消息层。

如果消息层无法到达（由于网络中断或消息层处于停机期间），则数据获取层还必须提供所需的安全性保障和故障恢复机制。

为了确保该层的安全性，它应该能够支持本地持久化的消息缓冲，这样，如果需要，并且当消息层再次可用时，消息可以从本地缓冲区中恢复。该模块还应该支持故障转移，如果其中一个数据获取进程失败，另一个进程将无缝接管，如图 2-4 所示。

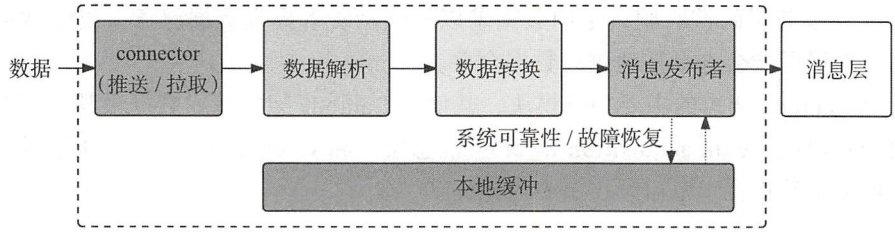


图 2-4 数据获取层组件设计

2.6.7 消息层——数据传输的保障

消息层其实就是数据湖架构里的消息中间件（Message Oriented Middleware，MOM），该层的主要作用是让数据湖各层组件之间解耦，同时保证消息传递的安全性。

为了确保消息能被正确传输到目的地，消息将会被持久化到某种存储设备中去。被选用的存储设备需要与消息处理需求匹配（结合消息大小及数量等因素）。更进一步来看，不论是读操作还是写操作，消息中间件都是按队列（queue）方式来处理的，队列天然适合处理串行存取，机械硬盘足以应付此类 I/O 操作。对于那些需要每秒处理百万级的消息的大型应用程序来说，SSD 能提供更好的 I/O 性能。

消息层组件必须能对消息队列进行入队列和出队列操作，如图 2-5 所示。对于大多数消息处理框架来说，入队列和出队列操作对应的是消息发布与消息消费。每个消息处理框架都提供了一系列库函数，用于与消息队列的资源连接（如 topic/queue）。

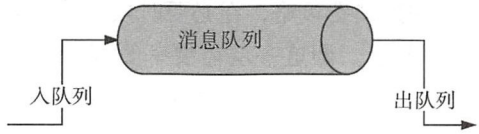


图 2-5　消息队列

任意消息中间件都支持两类与队列通信的方式以及 topic 消息结构，如下所列：

- 队列通常用于点对点（point-to-point）通信，每个消息应该只被某个消费者消费一次。
- topic 概念经常出现于发布 / 订阅机制中，在这里，一个消息被发布一次，但是被多个订阅者（消费者）消费。一条消息会被多次消费，但是每个消费者消费一次。在消息系统内部，topic 基于队列来构建；消息引擎（message engine）对这些队列进行差异化管理，以实现一个发布 / 订阅机制。

队列与 topic 都可以根据需要配置为持久化或非持久化。出于保障数据发布安全的目的，强烈建议将队列配置为持久化，这样消息将不会丢失。

从较高的层次来看，消息中间件可以抽象为由消息代理（message broker）、消息存储、topic/queue 等组件组成的框架或引擎。

图 2-6 中描述了消息中间件框架中的各种组件。请记住，图 2-6 给出的是一种较高层次的抽象，屏蔽了很多细节。这些组件稍后将在第 7 章中详细介绍。

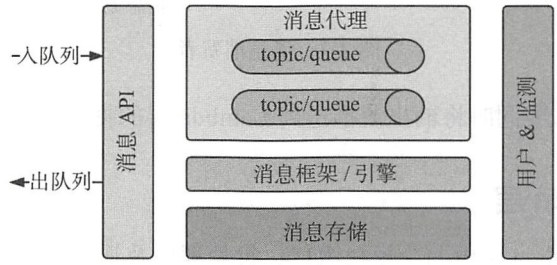


图 2-6　消息中间件框架

2.6.8　探索数据摄取层

数据摄取层负责消费消息层中的消息，对消息做适当的转换，从中提取所期望的信息，然后传输给 Lambda 层供其处理。数据摄取层的输出必须与期望的数据存储或处理格式一致。该层也必须保证消息以一致性的方式消费掉，即没有消息丢失并且每条消息至少被消费一次。

数据摄取层被期望能支持多个消费者 / 线程来并行消费消息。每个消费者必须是无状

态的，并且能快速处理流式数据。从消息层导出的多个数据流中的数据会源源不断地涌入Lambda层。数据摄取层必须确保消息消费速度不低于消息生成速度，这样消息/事件处理就不会有延迟。较慢的处理速度会导致消息层中消息的堆积，会对系统处理消息/事件的近实时特性造成伤害。该层应支持快速消费策略，在必要时恢复因消息堆积而导致的系统故障。

因此，该层有一个隐含的要求，即这一层需要一直保持近实时性，具有最小延迟，这样消息层就不会堆积任何消息。为了保障近实时性，该层必须有能力持续地消费消息/事件，及对故障进行恢复。

消息消费者扮演了向Lambda层递送消息供其处理的关键角色，如图2-7所示，因此消息消费者的内部组件与数据获取层非常相似，差别在于消息消费者知道从消息层（源）获取的消息及发送至Lambda层（目的地）的消息的格式。消息的消费行为可能是以微批量（micro-batches）方式来处理，这样能实现资源的最优利用，使系统效率更高。

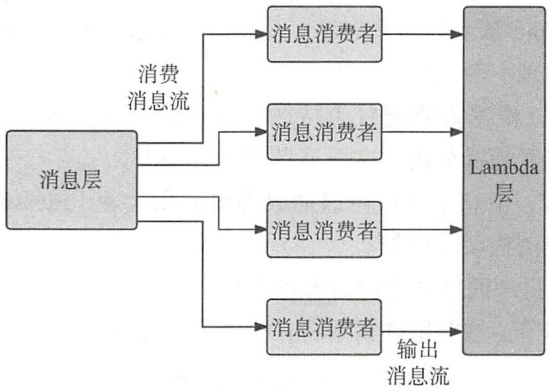


图2-7 消息消费者

消息消费者可能需要同时将输出流推送到Lambda层的批处理层及快速处理层。

2.6.9 探索Lambda层

前面提到过，Lambda层由两个模块组成：批处理层及近实时处理层（即快速处理层）。

1. 批处理层

批处理是处理海量数据的最传统方法之一，处理任务通常是长时间执行的。随着近期的各种大数据技术的诞生，批处理越来越高效，极大地缩减了处理时间。

一般来说，批处理对要消费的数据及要产生的输出结果非常了解。较早的时候，批处理的处理颗粒度很大，可能会在一次执行中处理整个数据集，也会用到多线程技术，并且提供了特殊的机制用来处理失效场景，以及执行一些操作流程来维护生产环境中的批处理作业。

Hadoop，作为一种大数据技术，因为其能满足批处理的各种要求而成为首选框架，它支持构建高效可扩展的批处理任务，这是传统批处理技术所无法比拟的。Hadoop中有两大

组件是执行批处理所需的，主要是并行处理及分布式存储组件。Hadoop 批处理已经被证明性能远优于传统批处理，原因如下：

1）使用经过优化的 MapReduce 范式的处理流程，执行速度很快。

2）顺序存储（sequential storage）有利于快速地顺序读 / 写。

3）存储备份机制（replicated storage）保证了数据的高可用性。

4）批处理任务执行期优先处理由作业调度器（job scheduler）管理的离得较近的数据。

基于 Hadoop 的批处理，由于具备了上面所述的这些能力，相较于传统批处理技术有巨大的改进。这里的分布式数据存储及并行处理由底层的 Hadoop 框架处理，Mapper 与 Reducer 任务更关注特定的数据处理。

MapReduce 处理范式并不是什么新概念，相反，它从大型机时代就已经出现在各种应用中了。该范式基于数据划分与规则，最早源自传统的多线程模型。其主要机制是将批处理分割为多个子任务，然后对这些处理任务的结果进行合并，最终产生整体输出，如图 2-8 所示。这样的话，多个任务并行执行且相互独立，各自处理各自的数据块（partition）。这样处理确保数据至少被处理一次，子任务的处理结果会被合并，并且结果已经去重了。由于 Hadoop 框架的内置功能，批处理的执行已经被证明是高度优化过的了，这使得 Hadoop 能处理主流的批处理问题。有了 Hadoop 及 MapReduce，从数据处理中推导出商业智能（business intelligence）成为可能，此外也可以集成更复杂的数据科学（data science）、机器学习（machine learning）方法，用于满足各种基于批处理的分析需求。然而这又会引发另外一个问题，我们如何满足实时数据处理的需求？

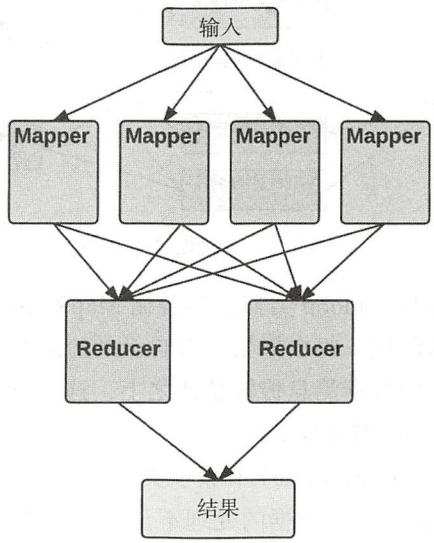


图 2-8　MapReduce 范式——批处理

有一种策略可以满足近实时数据处理的需求，那就是使用多种框架。设计这些框架就是为了解决近实时数据处理问题。Lambda 架构提供了多种机制使得可以在快速处理层中使

用这些框架。

早期人们尝试通过触发频繁的批处理操作来实现近实时处理，然而这种方式永远无法到达近实时处理的预期。

2. 快速处理层

快速处理层在 Lambda 架构中扮演近实时处理层的角色，在这里，消息 / 数据被快速摄取和处理，然后保存到存储层。

快速处理层主要满足数据近实时可用的要求，架构需要确保数据处理、存储、读取能达到近实时的预期。

为了保证近实时性，数据处理层、存储层、服务层必须保持对等的处理速度，这样才不至于在某个环节卡住。

已经存在一些流式处理的技术，较早的如 Flume，它可以与 HDFS 交互，部分解决了这个问题。然而，Flume 处理的是日志类型的数据，这些日志没经过进一步处理就直接存储到 HDFS 里了。数据的最终处理是在批处理中进行的，因此它本质上不是近实时的。

读者不难理解，依赖 Hadoop 的批处理，很难达到近实时处理的预期，因此需要一些另外的框架用于实现近实时处理，这些框架构成了 Lambda 架构的快速处理层。

最开始，这些框架是孤立的，它们没有与 Hadoop 生态系统很好地集成；随着应用的普及及功能的完备，显然这些框架需要与 Hadoop 生态系统集成以达到简化数据操作与管理的目标。图 2-9 显示了准实时处理流水线（pipcline）。

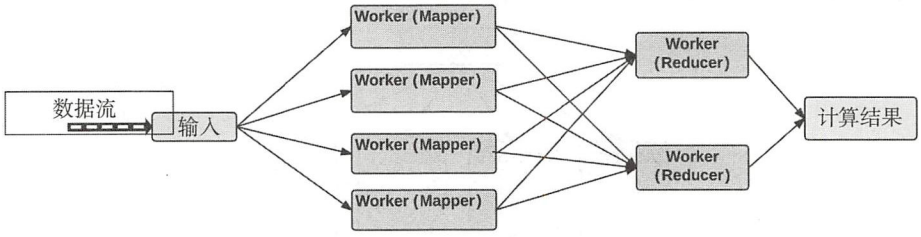


图 2-9 准实时处理流水线

这些框架的实现机制都与 Hadoop 类似，都遵循 MapReduce 范式。只不过它们在实现上针对近实时处理做了特殊优化。每个框架都有自己的流式数据处理及资源管理方式。它们中的大多数都构建在快速的内存消息传输功能之上，这是近实时处理中的一种有效的组件解耦方式，能将处理延迟时间降至最低。

实时处理通常依赖键 / 值（key/value）和索引类型的数据，因此需要一个强大的快速数据处理层，这样对键 / 值及索引类型数据的处理不至于对系统的实时性造成伤害。在这里，某些 NoSQL 技术扮演了重要的角色，在本章及后续章节中将会陆续介绍它们。

3. 服务层

服务层在 Lambda 架构中扮演着向消费者提供数据的角色。服务层应支持各种数据传输

协议。从消费者视角来看，这些协议可以分为数据推送和数据拉取两大类。

服务层必须能从数据湖的数据存储层消费数据，并且通过与消费者程序事先约定好的接口传输数据。

- 数据推送

任意向数据湖之外推送数据的机制被称为数据推送（data push）机制。有多种数据推送机制，下面列举了最常见的几种：

  - 数据导出（data export）：服务层必须提供数据导出所需的工具、控制及管理策略，为消费者应用程序导出与期望格式一致的数据。该层的部分功能与 ETL 类似，然而不同之处在于它主要是受消费者应用需求和使用方式驱动。在底层，可能是通过由服务层调度的批处理任务来对数据湖中的数据进行抽取、转换、加载等操作，最后将结果数据加载到目的地。服务层的此类功能可能使用了一些嵌入系统中的 ETL 工具来执行 ETL 处理。
  - 数据发布（data publish）：服务层也可以通过发布数据到消费者应用程序订阅的 topic/queue 的形式来向外推送数据。这种数据推送方式其实是点对点（point-to-point）的数据推送，遵循发布（publish）/ 订阅（subscribe）模型。因为这些向外推送的数据必须是可以发布的消息，因此消息越小越好，以便于提高系统性能。

- 数据拉取

任意支持数据消费者从数据湖向外拉取数据的机制被称为数据拉取（data pull）机制。下面我们来讨论几种最常见的数据拉取机制：

  - 服务（service）：数据传输最流行的机制之一是数据服务。这包括在数据湖上构建 Web 服务（REST / SOAP），这样可以通过服务将数据暴露给消费者应用程序。这对于通过 HTTP 传输的相对较少数据的消费行为来说，很容易满足其近实时要求。这也源于将数据作为服务的概念，在这里，所有数据在服务中已经准备好且为可用状态。服务请求和响应的定义必须简洁明了，以便于能保持足够的通用性以满足多个消费者的需求。这也隐式意味着必须对数据访问进行高度优化，以保证具有随机访问能力、秒级以内的响应时间及可对大数据集进行处理。这些服务更适用于只读数据服务，而不是可更新的数据服务。
  - 数据视图（data view）：数据湖也可能实现为基于数据视图的数据传输机制，各种应用程序可以连接到这些视图，然后拉取数据。这种服务数据的机制已经非常普遍，因为它结合了简单、易于维护和访问等优点。一旦数据通过数据视图对外公开，任何获得授权的消费者应用程序都可以通过标准驱动程序直接连接到这些数据视图，以及执行任何允许的数据处理。这些视图通常是物化视图（materialized view）以保障执行性能，并且隔离对性能有影响的查询，限定它们只能在特定的表上执行。然而，物化视图也需要进行刷新，可以以增量或彻底重建的方式进行，这个过程被称为刷新周期（refresh cycle）。如果要在一个刷新周期里彻底重建物化

视图，则需要有对应的控制机制，以保障刷新周期中的操作对数据服务没有影响。传统的做法是使用同义词（synonym）；在最近的一些技术中是通过数据集副本来实现同样的效果。

4. 数据存储层

Lambda 架构中的数据存储层必须提供灵活的访问机制，同时也应该对系统进行高度优化以满足批处理及近实时处理的需求。换句话说，存储层必须同时支持数据的顺序访问和随机访问。在典型的 Lambda 架构中，以下功能层级直接依赖数据存储层。

- 批处理层

从数据访问的角度来看，批处理过程需要对数据进行顺序访问，并且存储层应针对此类操作进行优化。Hadoop 是一种按块（block）读取和写入数据的技术，每个块都包含了序列数据中的一部分。甚至 Hadoop 中的块级访问都是按顺序进行的，以确保任何批处理在磁盘上的操作速度都足够快，即使是在旋转式的机械磁盘（普通硬件）上。

- 快速处理层

快速处理层需要以近实时的方式对其收到的数据消息进行操作，因此它必须能随机访问存储系统快速检索所需的信息，以及将处理后的数据快速写入存储系统。

- 服务层

服务层执行多种操作，它们对磁盘的顺序和随机访问都有需要，这依赖数据传输本身的特性。例如，如果需要在大数据集上执行数据导出，服务层通常会触发一个批处理进程来导出所需数据，此时主要依赖顺序数据访问。如果数据传输以数据服务的形式进行，所需的磁盘访问必须支持随机数据访问，以确保数据服务的响应时间长度低于预期。图 2-10 给出了数据访问模式示意。

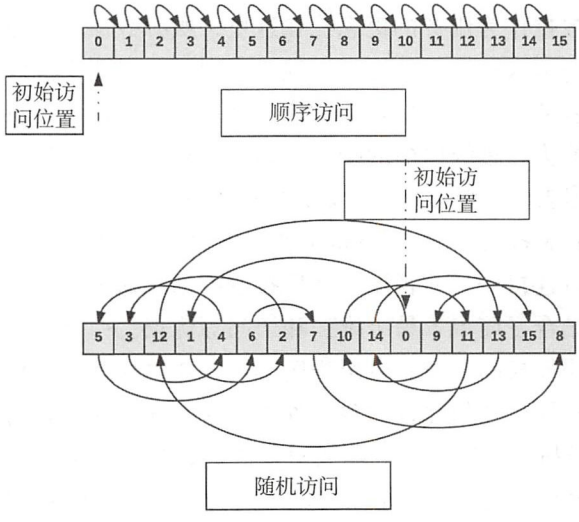


图 2-10 数据访问模式

因此，从数据访问的角度来看，数据湖中的数据可分为两大类：非索引数据（non-indexed data）和索引数据（indexed data）。

- 非索引数据：一般来说，原始数据进入数据湖中，会顺序、分块存储。这些数据块也构成了数据处理的单位。因为非索引数据是顺序存储的，因此它适用于批处理，并且输出数据也是顺序存储的。非索引数据是顺序数据，因此它对基于关键字的检索的支持非常有限（只有少量存储格式支持这种检索）。有些数据存储还支持某些级别的索引和分区（partitioning），以便能以最快速度定位到数据所在的位置。此类数据通常用于批处理，其目的是提高批处理的处理速度，将批处理进程迁移到数据附近，使得待处理数据的移动最小化，充分利用本地数据。这是 Hadoop 生态系统中 MapReduce 过程处理速度很快的关键原因之一。
- 索引数据：在维护数据湖中索引数据这个背景中，我们试图寻找能随机寻址和访问数据的解决方案。在支持随机数据的存储和访问模式方面，底层硬件扮演了一个至关重要的角色。正如前面内容曾经提到过的那样，SSD 非常适合这种场景。但是 SSD 也有它本身的问题，如成本、故障频率、数据容量、耗电量等方面的问题，这些问题又常常迫使企业不得不考虑使用机械硬盘。在这种背景下，需要对期望达到的 I/O 速率进行权衡。同时，实施者也可以考虑分级存储，例如，将高 I/O 数据或事务数据索引存储在 SSD 中，其他数据存储在机械硬盘中。

现今几乎所有数据索引框架都同时支持 SSD 和机械硬盘。这里介绍几种该领域领先的数据索引框架，它们在大数据技术及 Lambda 架构这些背景中得到了广泛的应用，分别是 Solr 和 Elastic（Elasticsearch）。这两个框架都基于 Lucene 引擎，Lucene 是一种开源的全文索引引擎，Solr 与 Elastic 的核心索引及检索能力均来自 Lucene，它们也都各自实现了一些额外的索引及检索功能。这两个框架都能保证在海量数据集上亚秒级响应，适用于快速处理层的快速数据检索和持久化。Elastic 与 Solr 都支持数据索引及快速检索。

- 基于存储的分类：当存储系统的差异更多的与访问模式相关时，数据存储可以根据存储机制来分类，如图 2-11 所示。

5. 关系数据存储

此类数据存储系统是过去几十年最流行的，关系数据是结构化数据，代表了实体（entity）间的关系。关系数据存储（relational data store）系统多年之前就非常成熟了，在企业间被广泛使用。直到几年前，这些数据存储系统还用于构建每个企业的主存储层，几乎所有的企业数据都保存在关系数据存储系统中，因为它们提供了一种非常合理的组织和管理数据的方式。

6. 分布式数据存储

虽然关系数据存储系统在处理关系数据库方面非常有效，但人们很快就意识到它们可能不适合处理其他类型数据的存储。其他类型的数据包括半结构化和非结构化数据。保持

关系数据存储的可扩展性使之能适应海量数据的存储与访问涉及复杂的处理和实践。这些挑战已由最近出现的一系列的分布式数据存储（distributed data store）（以分布式文件系统方式实现）和 NoSQL（Not only SQL）解决。Hadoop 已经成为现今最流行的分布式文件系统，虽然已经有一些 NoSQL 数据存储存在了，它们中的每一种只解决了一类特定问题。所有 NoSQL 数据库在分布式数据管理实现上基于类似的理念，不过它们可以进一步分为以下几类，如图 2-12 所示。

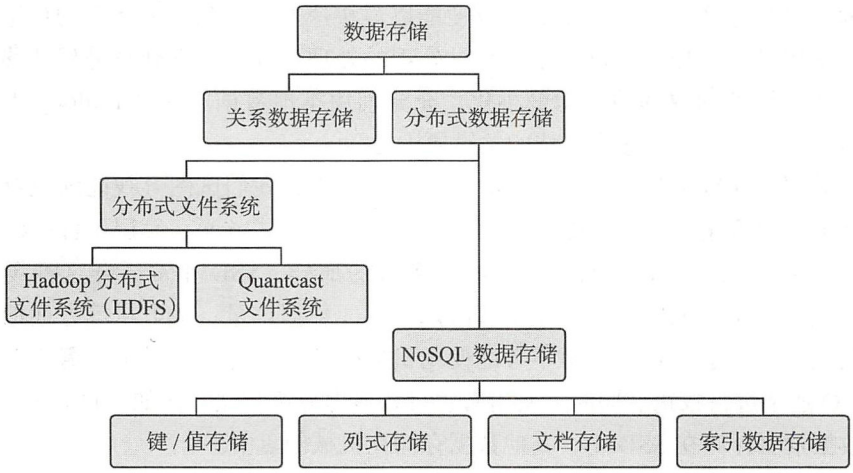


图 2-11 数据存储

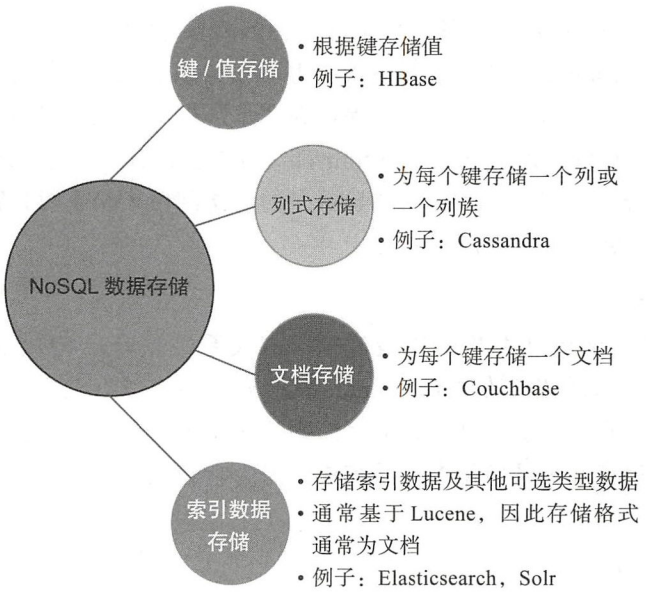


图 2-12 NoSQL 数据存储分类

图 2-12 中显示的每一类都解决了特定的数据访问及管理问题。

非卖品，仅供非商业用途或交流学习使用

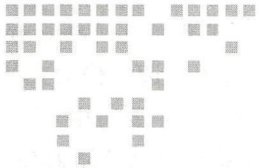
例如，键 / 值存储最适合处理时钟或机器数据，而可访问性需求则可以通过基于 key 的访问来完成。同样地，列式存储（columnar store）提供了非规范化（denormalized）的存储机制，其中数据存储为列或列族，而不是行，列式存储适用于读操作为主的场景，预计将加大对写操作为主的场景的支持。文档存储（document store）主要适用于将整个文档按 key 存储的场景。这些文档通常是 JSON 格式的，这些存储系统也可以存储 JSON 文件，并提供了 JSON 友好的查询引擎对外提供查询服务。当需要海量数据集上实现亚秒级查询，且以查询操作为主时，索引存储（index store）是首选。

上面介绍的每种数据存储都有大量的相关出版物，因为它们都具有大量独有的特性，适用于企业中海量数据处理和管理的各种场景。就数据湖而言，我们将在后面的章节中挑选若干种数据存储系统来展示数据存储层的多个侧面。在第 5 章中将介绍 HBase，在第 10 章中，将会介绍 Elasticsearch 如何作为 NoSQL 在数据湖中使用。

2.7　总结

在本章中，从较高的层次逐个介绍了数据湖及 Lambda 架构的组成部分，为本书后续章节的学习奠定了基础。之后将深入技术细节，介绍如何实现数据湖及 Lambda 架构。本章介绍了数据湖相关概念，以及某些关于数据获取层、消息层、数据摄取层以及 Lambda 架构层（快速处理层及批处理层）的高级概念。我们同时也在某种程度上讨论了数据存储及随机访问与随机访问之间的差异。

非卖品，仅供非商业用途或交流学习使用



Chapter 3 第 3 章

Lambda 架构：一种数据湖实现模式

在前一章中介绍数据湖的一系列概念时，粗略地提到了 Lambda 架构。在本章中，我们将介绍 Lambda 架构的一些细节，并解释该架构模式在本书的数据湖实现方案中的重要意义。

本章虽然会尽量涵盖 Lambda 架构范式的全部细节，但是并不会给出任何技术实现。这是为了保证读者能够首先理解这些模式的概念。我们在后续的章节中将详细介绍该模式的技术支撑。

本章中，读者将详细学习 Lambda 架构模式。了解了该模式之后，读者将会看到它是如何成为构建数据湖的一个不可或缺的组成部分的。

3.1 什么是 Lambda 架构

Lambda 架构不依赖于技术，不关心具体的技术实现，而是定义了一系列实用而完善的原则来处理大数据。Lambda 架构是一种非常通用的模式，它试图满足大多数大数据应用程序的共性需求。这一模式允许我们同时处理历史数据和实时数据。过去，我们使用两种完全不同的应用程序来分别处理事务类——联机事务处理（OnLine Transaction Processing，OLTP）数据和分析类——联机分析处理（OnLine Analytical Processing，OLAP）数据，但这两类程序不能混合在一起；它们独立运行，且相互之间不通信。

下列要素说明了什么是 Lambda 架构：

- 一套模式和准则。Lambda 架构定义了一套面向大数据应用的模式和准则。更重要的是，它允许同时查询历史数据和实时新增的数据，并且获得期望的分析视图。

非卖品，仅供非商业用途或交流学习使用

- 处理历史数据（批处理）和实时数据。
- 技术无关和通用性。Lambda 架构是一种通用的模式，完全不依赖于任何技术，而且任何技术只要能满足需求，都可以在 Lambda 架构中应用。
- Lambda 架构清楚地把责任划分到不同的功能模块 / 层中。它按照层来划分职责，完美地遵循了设计模式中的关注点分离原则。
- 领域无关。作为一种通用的模式，Lambda 架构可以应用于不同的业务领域。

3.2　Lambda 架构简史

Nathan Marz 创造了 Lambda Achitecture（LA）这个术语，用来描述一种通用的、可扩展的、容错的数据处理模式。他在 BackType 和 Twitter 上收集了大量关于大数据技术的专业知识。该模式是一种概念：通过使用两个重要组件来处理海量数据，这两个组件分别是批处理层和快速处理层。Nathan 把他的发现和经验概括为 Lambda 架构，该架构需要满足一些重要的架构设计原则，例如：

- 线性可扩展原则：它应该支持水平扩展（scale out）而不是垂直扩展（scale up），并且应该适用于不同类型的用例。
- 容错原则：它能够承担弹性较大的工作负载，还应能保护系统，免于受硬件故障、软件故障以及人为错误的影响。
- Backtype：读取和更新。
- 可扩展原则：易于管理，易于扩展，易于添加新特性和数据元素。

在网址 http://lambda-architecture.net/ 上有很多详细的文档。随着大数据的出现和企业对实时分析和数字化转型的关注，这一架构模式变得愈发重要。不过，为什么这一模式被命名为 Lambda（符号 λ）尚无明确的说法。

然而，我们一直认为这个希腊字母与这一模式相关联是因为数据来自两个地方。批量数据和快速的流式数据代表 Lambda 符号的弯曲部分，然后通过服务层（线段与曲线部分合并）合并，如图 3-1 所示。

图 3-1　Lambda 符号

3.3　Lambda 架构的原则

Nathan Marz 在《Big Data：Principles and best practices of scalable realtime data systems》一书中详细地介绍了 Lambda 架构模式。他提出来的架构模式主要基于以下 3 个原则，其中部分原则已经在上一节简要介绍过：

- 容错原则
  - 硬件
  - 软件

非卖品，仅供非商业用途或交流学习使用

- 人为
- 不可变数据（immutable data）原则
- 重新计算（re-computation）原则

下面将详细说明每一个原则。

3.3.1 容错原则

Lambda 架构模式中包括了容错能力，即处理硬件错误、软件错误或人工引起的错误的能力。大数据应用对容错有特殊的要求，而该模式迎合了大数据的要求，能保证系统不受数据丢失或数据损坏的影响。如果达不到这一要求，在大多数案例中，系统在遇到故障时是不可恢复的。因此该原则非常重要。

人工引起的错误是一种常见的错误。典型的就是日常操作中触发的操作性错误。另外一些常见的错误是引入生产系统的 bug，在系统长期运行以后会引发各种问题。因此系统需要经过精心设计，解决这些问题。

3.3.2 不可变数据原则

从各个源系统导入的数据应该按它们的原始数据格式来存储。更加重要的是，这些数据本质上是不可改变的。由于数据不可改变，那么就不会增加因人工而导致的错误，也在某种程度上减少了数据的丢失与损坏。该原则允许选择和插入，但是不允许更新和删除。为了保障数据处理速度和性能，数据是以非规范化范式来存储的。数据不可变性使系统变得简洁，更易于管理。

3.3.3 重新计算原则

由于数据湖中的原始数据一直处于可访问的状态，系统始终可以通过对原始数据进行重新计算来满足新的需求。此外，因为将数据绑定到特定模式会给重新计算带来问题，所以数据更倾向于无模式（schema-less）的存储。并且，将数据绑定到特定模式还会带来开发和维护方面的开销。

在以 Lambda 架构模式实现数据湖时，我们将看到这些原则是如何实现的。

3.4 Lambda 架构的组件

本书前面已经提及 Lambda 架构的各种组件，阅读这些章节之后，读者肯定对它们有所了解了。本节及后续小节将会详细介绍 Lambda 架构的各种组件。这里的介绍竭力避免涉及任何具体的技术，因为当讨论至具体的功能模块时，只要能满足模式的要求，就可以使用市面上任何合适的技术框架。理解每个功能模块及其担负的主要功能是非常有必要的，只有深刻理解这些，才能更有效地阅读后续章节。在数据湖背景中，Lambda 架构的组件只构

非卖品，仅供非商业用途或交流学习使用

成了其中的一个功能模块，即 Lambda 层。下面将带领读者逐一了解 Lambda 层的各个模块。Lambda 层的主要模块如下：

- 批处理层
- 快速处理层
- 服务层

Lambda 架构的示意图如图 3-2 所示，其中显示了该模式中的重要功能模块。

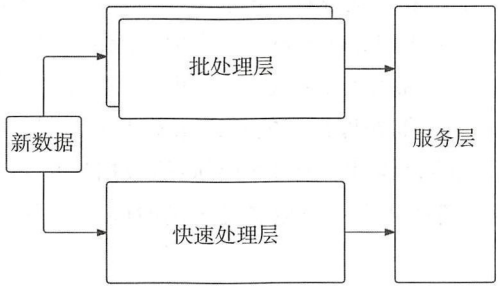


图 3-2　Lambda 架构中的组件

正如读者从图 3-2 中清晰地看到的那样，新到达的数据被同时发送到批处理层和快速处理层。批处理层持续地对每个时间间隔内的批量数据重新计算并生成视图。快速处理层同时也创建相应的实时视图 / 快速视图（speed view）。而服务层则对外部查询的处理做了精心编排，查询会同时被分发到批处理层及快速处理层，然后对结果进行合并，最后将查询结果返回给请求端。

一旦批处理视图被创建，则该视图不再考虑之后到达的新数据，这些新数据被用于构建新的实时视图。旧的批处理视图会一直保存和存档，何时被抛弃则依赖应用案例和技术实现。

处理大数据的新方法是按照图 3-3 所示的数据流水线的方式，其中数据是从真实来源以最原始的格式获取的。然后，我们基于原始数据创建适当的视图来适配业务需求；同时根据需要来使用这些视图。Lambda 架构的核心工作是通过批处理层和快速处理层产生适当的视图，然后服务层登场，充当用户查询与视图之间的桥梁。

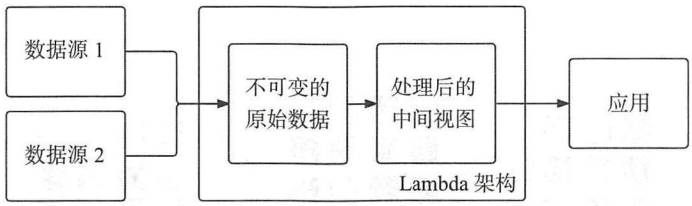


图 3-3　大数据流水线

3.4.1　批处理层

批处理层尽可能地按数据最原始的格式来存储数据。由于在存储的过程中不存在遗漏

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

或转换，因此，可以在不同的阶段从不同的维度衍生出许多不同的用例。在批处理层中，主数据以不可变状态存储，可以被访问也可以被用于各种分析。因为数据是不可变的，更改和删除是被禁止的操作。数据总是被附加上一个时间戳，以便在需要某些数据时，可以用版本最新的时间戳查询以获取最新记录。删除操作也是禁止的，因为很多分析都依赖这些被删除记录的细节。

在原始数据上执行查询会消耗大量的处理时间。为了避免因查询细节处理导致的延迟，系统按照一种周期性方式来处理数据，视图经过调校尽量接近所需结果的生成和存储格式，这种视图称为批处理视图（batch view）。当批处理视图需要重新生成时（上一次批处理之后，又有新数据进入了），那么旧的批处理视图会被抛弃掉。前面提到过，容错能力是Lambda架构的核心原则之一。批处理视图的每次重新生成，尽管是非常耗时的，也要能应对各种可能错误，保持系统的鲁棒性。有多种不同的数据处理方法可以缩减处理时间，相比之下传统的批处理一般要消耗几个小时或者几天的时间。如图3-4所示即为Lambda架构的批处理层示意。

图3-4 Lambda架构——批处理层

批处理层的持久化存储需要适应大数据量，同时应该支持大规模的随机读取。然而，它不需要支持随机写入，因为数据是按给定的频率批量加载的。

对于单一客户视图用例，图3-5显示了如何通过从客户的主数据集生成所谓的批处理视图（中间视图）来实现批处理层。

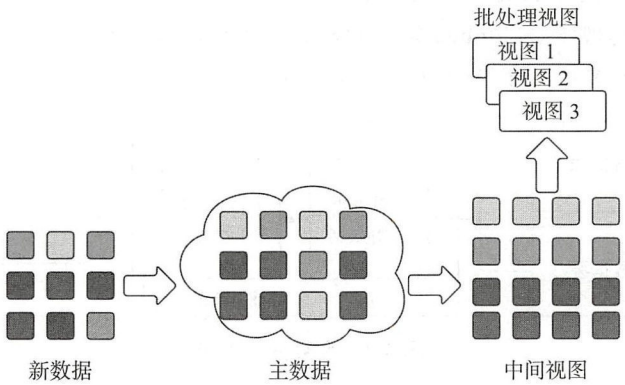


图3-5 单一客户视图——批处理层

对于单一客户视图的用例，客户数据流入批处理层，并在批处理层中维护主数据集；

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

然后以设定的批处理时间间隔来创建批处理视图。在需要时，服务层会查询这些视图，与快速视图合并，并将结果发送给用户。

3.4.2　快速处理层

快速处理层也称为实时层，是为了满足实时分析的需要。批处理层以指定的时间间隔运行，而在一个批处理运行完成和另一个批处理开始之间，业务用户仍然希望对数据进行分析。快速处理层负责合并批处理视图和实时数据。现在，批处理窗口可以减小，但是由于批量处理大量数据，批处理层的处理通常较为耗时，但业务不能因等待批处理层的延迟而滞后。为了实现近实时数据分析，增量数据以低延迟的方式进入快速处理层。一旦批处理层某次执行完毕，并覆盖了快速处理层中的数据，那么实时视图就会被抛弃，然后系统进入下一轮处理。

用户提交查询时，会同时向批处理层和快速处理层进行查询，然后将结果合并，并以用户期望的方式将查询结果发送给用户。

为了实现容错性和能够从错误中恢复，批处理层和快速处理层在任何时刻都可以重新计算（从原始数据恢复批处理层），回滚（恢复到批处理层的前一个状态）或刷新（为快速处理层重新生成视图）。

快速处理层有几个非常重要的概念，为了实现该层的基本目标，必须了解它们：

- 增量计算（incremental computation）：使用现有的实时视图和新增数据来创建新的实时视图。如前文所述，这样做是为了减少时间开销，使得数据可以接近实时地分析。
- 最终一致性（eventual consistency）：某些实时计算任务非常复杂，而且很耗时。此时系统应该追求快速得到问题的近似解（接近正确答案）。之后在某个时间点（通常不会延迟很久），结果变得完全正确。

快速处理层的存储需要支持随机读取和写入，以满足增量更新的需要。快速处理层确实允许数据快速变化，但与批处理层处理的大量数据（时间跨度为数月或数年）相比，快速处理层处理的数据集非常小（比如每次处理一天的数据）。

按照上一节所述的通用方法，这里还创建了一个所谓的快速视图（中间视图）来满足要求，如图 3-6 所示。

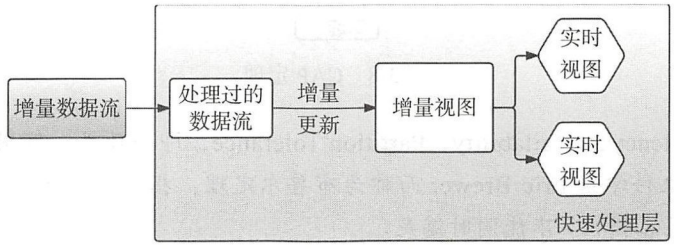


图 3-6　Lambda 架构——快速处理层

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

就单一客户视图而言，快速处理层的实现方式如图 3-7 所示。

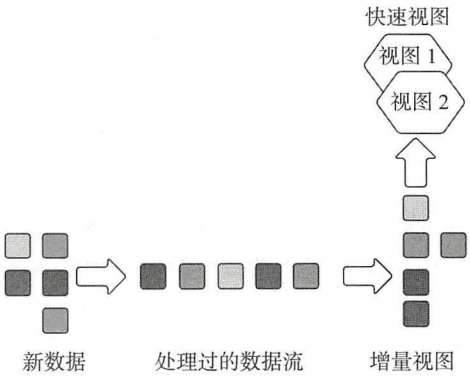


图 3-7 单一客户视图——快速处理层

当新数据流入快速处理层，新增的数据在这里被处理并构建出合适的快速视图。当服务层有需求时，快速视图会与批处理视图合并，然后结果被返回给用户。

1. CAP 定理

前面我们确实探讨了一些非常重要的方面（比如最终一致性，下面将深入探讨），也简要介绍了其他方面。在更详细地解释最终一致性之前，需要解释一个非常重要的定理——CAP 定理（见图 3-8）。

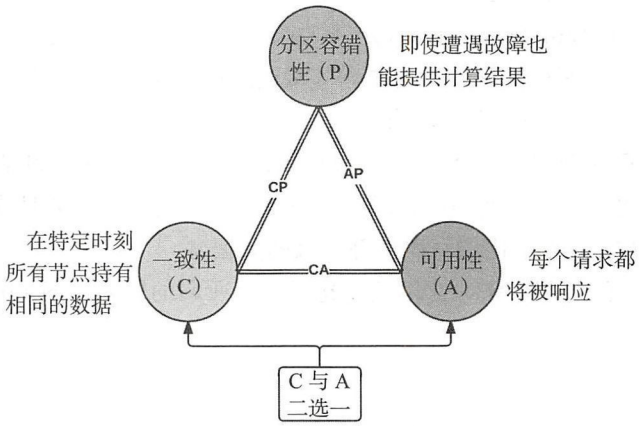


图 3-8 CAP 定理

CAP（Consistency，Availability，Partition Tolerance，即一致性、可用性、分区容错性）定理，也因计算机科学家 Eric Brewer 而称为布鲁尔定理，指出在分布式系统中三者（一致性、可用性、分区容错性）不能同时满足。

——维基百科

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

在 3 个特性中，分布式系统在数据分块时只能满足 C（一致性）或 A（可用性）其中之一。分布式系统必然会出现网络故障，在这种情况下，只能接受网络分区。表 3-1 中会简要说明这 3 个重要方面。

表 3-1　一致性、可用性、分区容错性说明

特　性	说　明
一致性	• 当数据分区存储时（分布式存储），所有节点在特定时间点看到的数据都应该是一样的，并且应该在任意时刻都保持这种一致性 • 查询时，每个节点将返回最新的数据。否则，系统将报错
可用性	• 在任何时候，系统会为接收到的每个请求生成一个有效的响应 • 这样做并不意味着每个请求的响应中都会包含最新的信息（数据）
分区容错性	• 即使发生网络故障或数据丢失，系统也能够继续运行

在数据湖背景中，Lambda 架构的实现也采用了 CAP 定理。通常在这样的环境中，可用性与一致性难以兼顾。基于这方面的原因，数据的一致性最终会得以实现，不过通常情况下，数据是近似一致的。这就是所谓的最终一致性。我们将在接下来的部分中详细介绍这方面的内容。

传统的满足 ACID（Atomicity，Consistency，Isolation，Durability，即原子性、一致性、隔离性、持久性）要求的存储系统，比如 RDBMS，选择了一致性和可用性。而 NoSQL 存储系统，由于它们的特征更适合数据湖，遵循了 BASE（Basically Available，Soft State，Eventual Consistency，即基本可用、软状态、最终一致性）思想，并选择了可用性而不是一致性。

2. 最终一致性

最终一致性是分布式计算中使用的一致性模型，用来实现高可用性，基本保证如果对给定数据项不做任何更新，则最终对该数据项的所有访问都将返回最后更新的值。

——维基百科

至此，读者可以清楚地了解该保证背后的意义，即一致性与可用性之间的较量。

分布式系统中的数据期望最终保持一致，并随时保持可用状态。对于数据湖来说，保证可用性比保证一致性更重要，但这完全取决于用例。我们不能向最终用户展示或使用不良数据，因为这会对公司品牌产生巨大冲击。所以，用例需要经过仔细的验证，读者需要将这一条作为数据湖实践的通用原则。

在数据湖（Lambda 层）背景中，即使快速处理层受损（数据丢失或数据损坏），最终批处理层和服务层也将纠正这些错误，并通过服务层正常响应查询并返回结果。Lambda 架构中的快速处理层只会将数据保留一段时间，这样批处理层和服务层可以重新开始处理数据并确保一致性，一旦完成，快速处理层的视图将被丢弃。

3.4.3　服务层

服务层的核心任务是响应查询，将批处理层和快速处理层创建的视图展示给用户或其

仅供非商业用途或交流学习使用

他系统。

除此之外，这一层还有很多精细的工作需要进行。快速处理层需要一直关注批处理层，以观察是否完成了必要的批处理操作。这样一来，在批处理视图中进行批处理操作后，需要立即更新批处理层的存储。紧接着，当前的快速处理层视图将被丢弃，这是一种簿记行为，用以保持快速处理层存储大小在控制范围内。

观察图 3-9，可以看到批处理层和服务层是如何工作的。

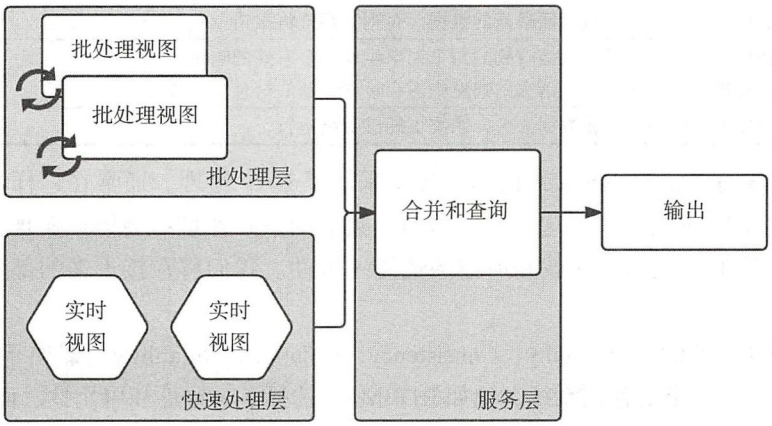


图 3-9 Lambda 架构——服务层

当查询到达服务层时，服务层分别向批处理层和快速处理层分发查询。查询执行时会查看记录的时间戳，并根据提供的参数获取结果。一旦从这两层获得查询结果，服务层合并这些记录并使用各种业务逻辑进行计算，然后以系统或用户需要的格式产生最终输出。基于数据质量的因素，批处理层更加可靠（每个批处理操作完成后会进行下一轮重新计算），如果发生冲突，批处理层的视图会覆盖快速处理层的视图。

3.5 Lambda 架构的完整工作原理

图 3-10 中显示了 Lambda 架构的完整工作原理。

正如前文中简要描述的那样，主数据集在批处理层中进行维护和管理。新数据到达时，会被同时分派到批处理层和快速处理层。一旦数据到达批处理层，按照常规批处理时间间隔，每次都从头开始重新计算并生成批处理视图。类似地，只要新数据到达快速处理层，快速处理层就会使用新数据生成快速视图。在查询到达服务层时，它会合并快速视图和批处理视图来生成适当的查询结果。

生成批处理视图后，快速视图将被丢弃，除非有新数据抵达，否则只需要查询批处理视图，因为此时批处理层中拥有所有的数据。

表 3-2 中列出了 Lambda 架构中批处理层、快速处理层和服务层的作用及特点。

仅供非商业用途或交流学习使用

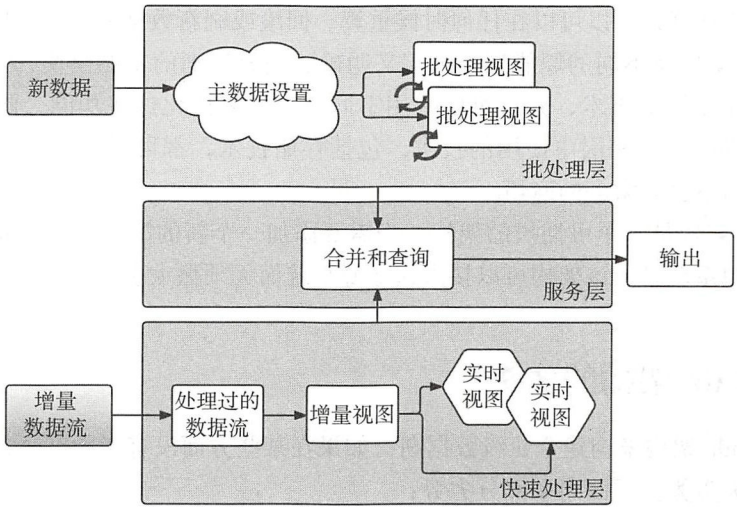


图 3-10　Lambda 架构的完整工作机制

表 3-2　批处理层、快速处理层、服务层的作用及特点

Lambda 架构	作用及特点
批处理层	• 存储不可变的数据 • 数据量不断增长 • 随时重新计算视图
快速处理层	• 不间断的数据流 • 存储可变的数据 • 数据量较小 • 视图只在指定的时间段内存在并且每过一段时间就会被丢弃
服务层	• 负责索引并确保对外暴露的批处理视图工作良好 • 对外暴露由快速处理层增量创建的实时视图 • 将批处理层和快速处理层的视图合并统一

3.6　Lambda 架构的优势

因为 Lambda 架构有很多优势，所以我们选择使用它来为企业构建数据湖。Lambda 架构的典型优势如下：

- 数据以原始格式存储。因此，只需要创建新的批处理视图和快速视图，就可以随时将新的算法、分析方法或者业务用例应用于数据湖。传统数据仓库最大的优点之一是数据会被清洗和存储。正因为这一点，新的用例需要更改数据模式，这通常很耗费时间和精力。
- Lambda 架构的重要原则中包括了重新计算，这有助于纠正错误，而不会有太大的额外开销。随着越来越多的数据进入数据湖，数据丢失和损坏的代价难以承受。因为

仅供非商业用途或交流学习使用

可以重新计算，所以可以在任何时候重算、回滚或刷新数据来纠正这些错误。

- Lambda 架构将不同的职能划分为定义明确、边界清晰的功能模块 / 层。这样各层就可以使用不同的技术，并且可以采用不同的方法来优化这些功能，同时对其他层没有任何影响。这些层中使用的技术，包括存储技术，都可以随时替换为更好的解决方案，而不会有太多的麻烦。
- Lambda 架构是一个可插拔的架构。当需要添加一个新的源数据类型或者已经接入已有类型源系统时，该架构可以接入这些源系统而无须做太多改变。

3.7 Lambda 架构的劣势

选择 Lambda 架构来构建企业级数据湖，如果在某些方面没有考虑充分，确实会引入一些该架构的先天劣势。下面列举部分劣势：

- 由于包含几个不同的层，Lambda 架构通常被认为很复杂。必须经过充分的思考和操作以及付出一定的代价来保持各层之间的数据同步。
- 由于批处理层和快速处理层都是分布式的，且实现机制截然不同，维护和支持起来相当困难。
- 要构建基于 Lambda 架构的数据湖，必须掌握大量的技术。招聘掌握这些技术的人并非易事。
- 用开源的技术来实现 Lambda 架构并部署在云环境中并不容易。为了避免这种情况，使用云计算技术可以很好地实现 Lambda 架构，但是这样做，企业等于是把自己和某个云计算供应商绑定到一起，这被认为是一种先天的不足。
- 尽管这种架构模式已经存在了相当长的一段时间，所用到的这些工具仍处于快速发展和演化中，还不够成熟。不过，随着云计算的加速发展和创新，很快就会出现成熟的解决方案和工具。
- 目前，持续集成（CI）/ 持续交付（CD）已经是一种平常的要求。跟该领域的其他工具类似，持续集成 / 持续交付的工具也不成熟，所以很难支持太多自动化操作。
- 系统假设可能需要大量的硬件组件。从技术上讲，可以使用低端硬件组件，但是对于企业级用户来说，供应商通常会提供高端产品，这是企业和供应商之间长期以来约定成俗的供应模式。
- 相同的工作要实现两次（在批处理层和快速处理层），这也是 Lambda 架构模式一直被诟病的一点。

3.8 Lambda 架构技术概览

如同前文简要介绍的那样，Lambda 架构是有定义良好的原则的模式，而且是与技术无

仅供非商业用途或交流学习使用

关的。具体到各个组件 / 层，我们可以引入任何技术来完成所需工作。

随着各种云计算供应商的出现，甚至可以在云环境中获得现成的组件（许多是依赖云环境的），更甚者，这些组件已经实现了 Lambda 架构。在本书中，我们将创建一个实际的数据湖，其中 Lambda 模式只包含一个层，叫作 Lambda 层。

由于有非常多的技术可供选择，后面每章中技术框架的选择会让人感觉比较主观。在每层中选择一种新技术时，我们都会给出选择的理由，同时也对其他框架保持开放。我们还将简略介绍类似技术框架，以便在需要时读者可以自行进行替换。话虽如此，但实际上我们使用选定的技术框架来实现数据湖。本书的第二部分将详细介绍每一项技术，然后在第三部分中，这些技术会被配套使用，用于构建数据湖。

3.9　应用 Lambda

企业级数据湖是 Lambda 架构模式的应用案例之一。本书将更为详细地讨论这个话题。事实上，也有其他用例应用该模式，本节也会尝试尽量涵盖这些案例。

3.9.1　企业级日志分析

该模式的一个非常常见的用例是日志的获取及分析。ELK（Elasticsearch，Logstash，Kibana）技术栈在该领域比较领先，但是 Lambda 模式也可以很好地胜任该任务。日志可以是传统的应用程序日志，也可以是各种软件或硬件组件生成的各种日志。当需要企业级的日志管理和分析能力时，Lambda 模式的确是一个不错的选择。这些日志的量很大，生成速度也很快。同时，这些日志都是不可变的，而且确实需要提供命令以便分析师（应用程序开发人员和安全数据科学家可以利用这些数据来发现安全威胁）使用这些日志。

使用 Lambda 层中重要的层（接下来的内容中将详细介绍批处理层和快速处理层）来校验这些实时日志，同时也可以帮助理解过去的日志，以提供实时的洞察力。研发团队可以根据分析结果采取主动的行为。例如，应用程序的开发人员可以发现程序错误，而安全分析人员可以评估安全方面的威胁。

除了用日志分析来检测某些异常情况之处，还可以使用这些数据来做分析和审计。例如，在网站运营的场景中，Google Analytics 的数据会导入数据湖中，并用来发现一些趋势，这将有助于改善网站运营状况。

3.9.2　获取和分析传感器数据

随着物联网（Internet of Things，IoT）应用的日益增加，企业使用的传感器也越来越多。这些日渐增多的传感器产生了海量的数据，要从中提取出有用的信息却非常棘手。使用具有 Lambda 层的数据湖可以为用户提供一个可扩展的解决方案，以便对这些海量的传感器数据进行实时分析。

仅供非商业用途或交流学习使用

3.9.3 电子邮件平台实时统计

对企业来说，电子邮件平台现在起着非常重要的作用。因拥有大量来自业务应用和社交平台的数据，电子邮件营销（E-Mail marketing）平台的作用也举足轻重。很多电子邮件平台都可以跟踪每一封邮件的状态，比如打开、链接点击等。利用这些海量的跟踪数据，将其作为实时统计数据提炼出来，会有助于营销团队更好地选择目标客户并提供差异化的邮件内容。目前，主流电子邮件平台已经具备了对内容进行 A/B 测试的功能，这部分测试数据非常重要，营销团队可以根据它更好地准备营销邮件的内容。

3.9.4 实时赛事分析

很多比赛中可以利用历史的批处理视图和当前比赛的实时数据向观众呈现多维度的统计和分析。

批处理层可以持续进行批量数据处理，准备好各种批处理视图，将这些视图与当前的比赛结合起来，可以推测和展示当前比赛中运动员取得的成绩。以板球为例，在比赛进行时，球员可能取得不同的成绩。比如在测试中取得了 1000 分，这个成绩就可以在比赛时呈现给观众。

3.9.5 推荐引擎

各种商业推荐引擎同时使用了来自各种业务应用的历史数据及当前的实时数据，大多数时候是终端用户在网站上的行为数据。

3.9.6 安全威胁分析

日志可以从各种硬件和软件的组件中（包括业务应用）收集，把这些日志与过去的数据集进行对比可以用来分析安全方面的威胁。例如，根据分析的情况，可以采取不同的认证机制。比如说，可以把认证机制加强到双因子认证；如果在几分钟内反复收到来自某地的请求，就可阻塞这些请求并提交给人工审查，以决定是否授权执行当前事务。

3.9.7 多渠道用户行为分析

Lambda 架构也可以用来进行多渠道用户行为分析。比如分析用户的购物行为、过去的消费情况、社交行为等，然后用这些数据来进行精准营销。同时，基于对各种 A/B 测试的结果进行分析，营销团队就能采取恰当的方式来投放广告邮件。

3.10 Lambda 架构运行范例

Lambda 架构已经在业界取得很多成功的应用，下面是其中一些案例：

- Twitter：例如采用改良版的 Lambda 架构对推文进行情感分析。

- Groupon：多类企业应用。
- Crashlytics：使用 Lambda 架构的批处理层和快速处理层进行有效的移动端数据分析，产生有意义的分析结论。
- Stack Overflow：一个知名的问答论坛，拥有庞大的用户社区和丰富的活动。Lambda 架构被用来创建一个向登录用户推荐问题的板块。还有很多其他的数据分析，比如用批处理视图来分析投票数据。
- Flickr Magic View：修订版的 Lambda 架构通过结合批量计算和实时计算来创建一个神奇的视图（可参考 code.flickr.com）。

3.11　Kappa 架构

在本书中使用 Lambda 架构来构建数据湖的一个主要层（即 Lambda 层）。然而，读者也需要了解另一个简化版的 Lambda 架构，这个被热议的架构叫作 Kappa 架构。它和 Lambda 架构有着或多或少的相似之处，只是出于简化考虑，去掉了批处理层，只保留了快速处理层。其主要思想是避免从头开始进行批处理层计算，尝试把这些计算完全放在实时计算或快速处理层。如前文所述，Lambda 架构的一个缺点是必须编码并运行同样的逻辑两次，但 Kappa 架构避免了这个问题。

一图胜千言，图 3-11 中将 Kappa 架构和 Lambda 架构放在一起进行比较，读者可以清楚地看到在 Kappa 架构中唯一缺失的部分是非常重要的批处理层。

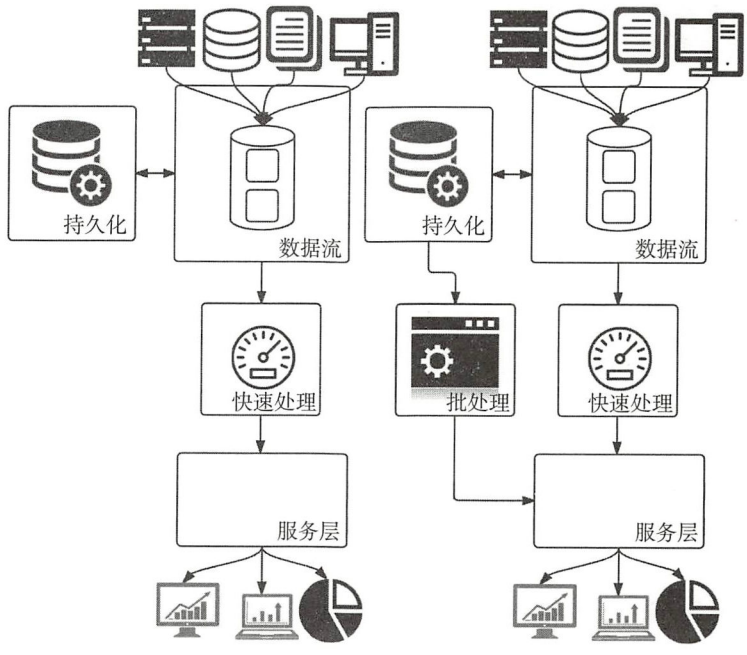


图 3-11　Kappa 架构（左）与 Lambda 架构（右）比较

我们认为，即使非常复杂，Lambda 架构仍然是构建数据湖的必经之路。同时也希望通过图 3-11 为读者提供另一个视角，让读者能够深入到实现方案中去，这就是为什么要简单地介绍 Kappa 架构。

Lambda 架构中存在一个批量处理层是有原因的，为了解决这方面的问题，Kappa 引入了新的技术框架，这些技术框架能够对大量数据进行相关处理。我们认为，在 Kappa 架构中，Lambda 架构中的批量处理层和快速处理层被融合在了一起。实际上，我们也可以在 Lambda 架构中使用这些新技术，并仍然保持层次分离和责任清晰划分。

3.12 总结

在本章中，详细介绍了 Lambda 架构。在本书的数据湖实现方案中，Lambda 层（Lambda 架构的一种实现）是不可或缺的一部分。

本章只介绍了理论层面的 Lambda 架构，并未涉及具体技术。希望读者能明白，在该模式中，任何技术都能用来实现该架构各个层的功能。下一章开始介绍各种技术以及这些技术的适用场景。

本章前几节对 Lambda 架构进行了简要介绍，后几节详细介绍了构成 Lambda 架构的多个功能模块 / 层。同时，还阐述了该模式的优缺点，并全面而详尽地描述了这个模式。

希望读者现在已经对数据、数据湖和 Lambda 架构的背景知识有了充分的了解。读者在下一章将会了解如何为企业实现数据湖。

第 4 章 Chapter 4

数据湖中的 Lambda 应用

前面几章中已经介绍过了，大数据可以用 4V 来定义，它们分别是 Variety（多样性）、Velocity（处理速度）、Volume（数据量）、Veracity（精准性）。同时也介绍了 Lambda 架构，以及它是如何合并两种截然不同的数据处理流水线的输出的。为了利用大数据技术解决各种数据处理问题，结合 Lambda 架构使用这些大数据技术可能是最好的办法，因为能同时收获两者的优点。大数据可以用 4V 来描述，尽管它指的是涵盖数据处理、信息管理的环环紧扣的整套解决方案，不过在大多数时候，Hadoop 框架几乎是大数据的代名词。虽然 Hadoop 最初实现是由 Apache 开源社区发起的，但它的即时需求带来了大量的商业支持。在一段时间内，社区见证了一些定制 Hadoop 版本的发布。如今最流行的商业版本有 Cloudera、Hortonworks、MapR 等。大家都知道，Hadoop 作为一个成熟的框架，最初由雅虎实现。最开始作为企业内部大数据处理工具，稍后遵从 Apache 开源协议开放源代码。此后 Hortonworks 与雅虎剥离，并继续维护该商业版本，与 Cloudera 和 MapR 等竞争。本章中将对 Hadoop 的相关技术进行概述，在概念上描述它们是如何帮助我们实现数据湖的。

在这里，我们想根据大数据的总体面貌和本书中为即将开始的章节选择的具体技术来预设一些场景。本书尽可能参考标准的开源发行版，这样示例和概念不用依赖具体的版本，尽管读者可以选择在任意发行版上运行。因此，这本书的内容将更加倾向于开源发行版。

4.1 Hadoop 发行版本介绍

大数据生态系统由多种功能构成，其中的每种功能都存在不止一种框架。对于每一个

具体的发行版，都会按照其独特的思路来设计实现，并会在某个方面具有竞争对手难以比拟的优势。

图 4-1 展示的是 Hadoop 框架中占市场主导地位的一些发行版，其中 Cloudera、Hortonworks、MapR 是商业发行版，而 Apache Hadoop 是开源版本。与开源的 Hadoop 框架相比，商业发行版具有很多自己的特性，如图 4-2 所示。

Cloudera MapR

Hortonworks

图 4-1 各种 Hadoop 版本

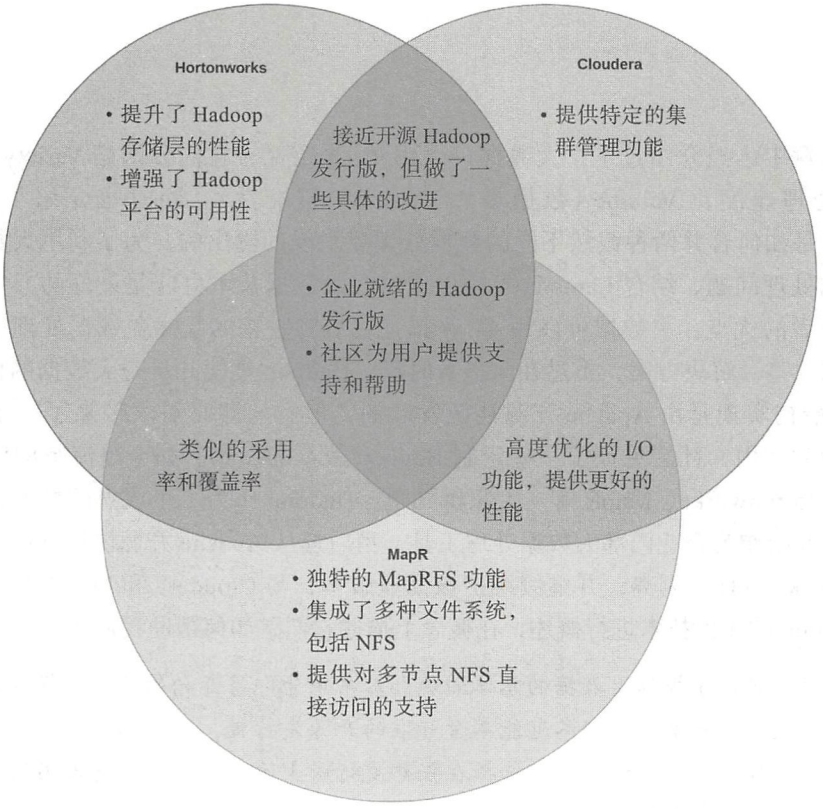


图 4-2 Hadoop 发行版及其特性

读者也许会有疑问，既然存在这么多种 Hadoop 发行版本，那么如何进行选择呢？遗憾的是没有一个标准答案。用户只能基于多个维度的特性来对它们进行比较判断，做出适合应用场景的选择。

非卖品，仅供非商业用途或交流学习使用

4.2　影响企业大数据技术栈选择的因素

对于任意企业，如果它希望采用特定的 Hadoop 商业发行版本，那么就应该考虑框架成熟度、使用方式等因素。下面将简单地介绍这些选择因素。

4.2.1　技术能力

每个发行版都有自己独特的功能，但是它们的大多数功能都是相似的。如果关注细枝末节，总是可以列出一个很大的功能列表，但是我们应该将重点放在一些最重要的功能上，以便进行比较和评估。

4.2.2　是否易于部署和维护

许多 Hadoop 发行版采用了通用的核心组件，但是在部署和维护方面做了很多定制。它们在自动化、可选监控接口、系统更新等方面有较大差别。

4.2.3　集成准备

很多发行版提供了与其他数据处理系统集成相关的多种能力，包括 Hadoop 的上游系统（inbound）或下游系统（outbound）。有些时候，这将成为选择特定 Hadoop 发行版的一个关键因素。

Apache Hadoop 作为一款伟大的大数据处理框架，企业在使用它时是否能得到专业支持和服务是非常关键的。基于这个目的，企业需要评估具体的 Hadoop 发行版本是否适与企业匹配，因为每个企业都有特定的流程、标准、监控方式、使用预期，换句话说，就是每个企业都有其特定的技术栈。本书的意图是从中立的角度来帮助读者理解如何利用 Lambda 架构构建数据湖，因此本书中将采用 Apache Hadoop 版本来介绍相关知识及演示范例。

4.3　批处理层与数据处理

Hadoop 技术的核心是它能够快速、高效、优化地执行批处理。它已经取得了空前的成功，被证明能有效解决部分企业内需要长时间运行的复杂批处理问题。Hadoop 最初的实现基于开源的 Hadoop 版本。然而，用户希望能得到专业的技术支持，因此很多特性被吸收到 Hadoop 中，这些特性能为企业提供配置、管理、监控、预警等方面的便利。这导致了很多进一步定制的 Hadoop 发行版的诞生，常见的发行版有 MapR、Cloudera、Hortonworks 等。

如图 4-3 所示，Hadoop1 框架被分为两大模块：存储及处理模块。与存储对应的是 Hadoop 分布式文件系统（Hadoop Distributed File System，HDFS），与数据处理对应的是并行计算框架 MapReduce。在 Hadoop2 中，随着 YARN（Yet Another Resource Negotiator）的出现，引入了很多改进的功能，如图 4-4 所示。

非卖品，仅供非商业用途或交流学习使用

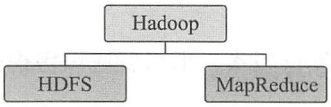


图 4-3 Hadoop1 框架

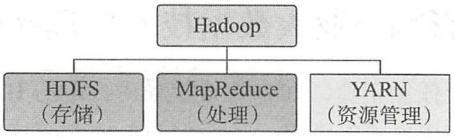


图 4-4 Hadoop2 框架

还有大量其他的值得考虑的框架，如 Pig、Hive、Impala 等。然而这些框架在其内部都是在执行 MapReduce 批处理作业（batch job），这些任务的执行和协调可以通过配置 Oozie 工作流中的 action、stage、parameter 来精准控制。

现在我们进一步了解 Hadoop MapReduce 批处理作业是如何工作的。Hadoop MapReduce 由若干核心组件构成，它们在最初的 Hadoop 框架中就已经出现了。Hadoop 已经从 Hadoop1 演进到 Hadoop2。Hadoop1 存在单点故障（single point of failure）的劣势。Hadoop2 中引入了其他关键特性帮助系统消除了单点故障。从讨论的角度来看，本书将覆盖 Hadoop2，并将它与 Hadoop1 进行深入比较。Hadoop2 的批处理框架由图 4-5 所示的关键组件构成。

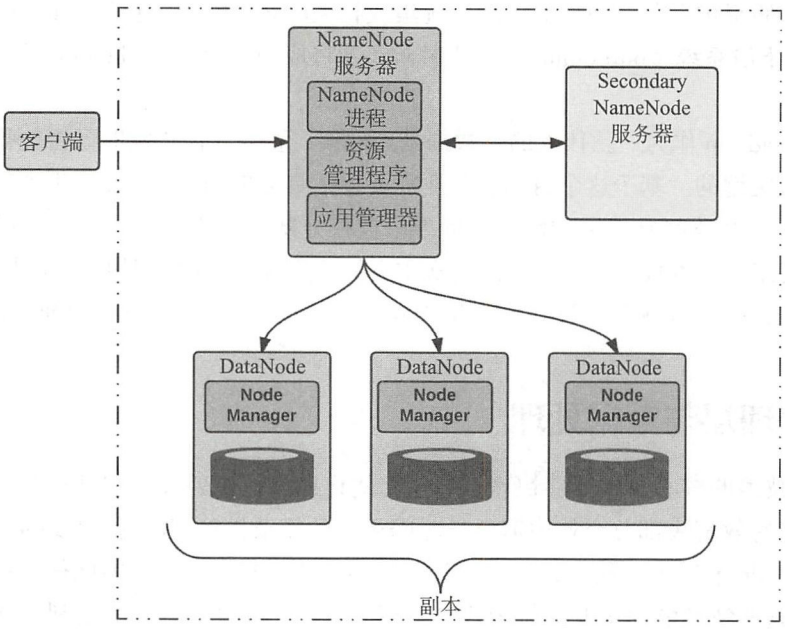


图 4-5 典型的 Hadoop 集群

4.3.1 NameNode 服务器

该组件负责调度和追踪散布在集群中的作业（job）。在 Hadoop1 中，它会导致单点故

非卖品，仅供非商业用途或交流学习使用

障，在 Hadoop2 中，该组件的功能被委托给 YARN 了，YARN 会在集群中创建多个 Job Tracker，这能保障系统避免陷入单点故障的困境。

4.3.2　Secondary NameNode 服务器

该组件用于记录 Hadoop 集群的内存快照，追踪预写式日志（write-ahead log）及事务属性。它通常用于备份集群状态。请记住，它并不是 NameNode 服务器的备份节点，只是用来记录快照。

4.3.3　YARN

YARN 在 Hadoop2 中被引入，它是整个 Hadoop 集群主要的资源管理者。此时相当于 YARN 为 Hadoop 集群创建多个 JobTracker，解决了 Hadoop1 的单点故障问题，因为某个 JobTracker 失效不会影响其他任务，这样，可以通过重新调度恢复单点故障对应的作业。

4.3.4　数据存储节点

数据存储节点（DataNode）在 Hadoop 集群中主要扮演存储数据的角色，作业实际上作为任务（task）在这些节点上执行。任务按如下方式调度：批处理作业处理向数据迁移，将任务分配给最有可能处理数据的节点。从执行的角度来看，这保证了批处理作业是经过优化的并且因就近处理数据带来了较快的性能。

图 4-6 揭示了典型的 Hadoop 批处理过程的内部工作细节。

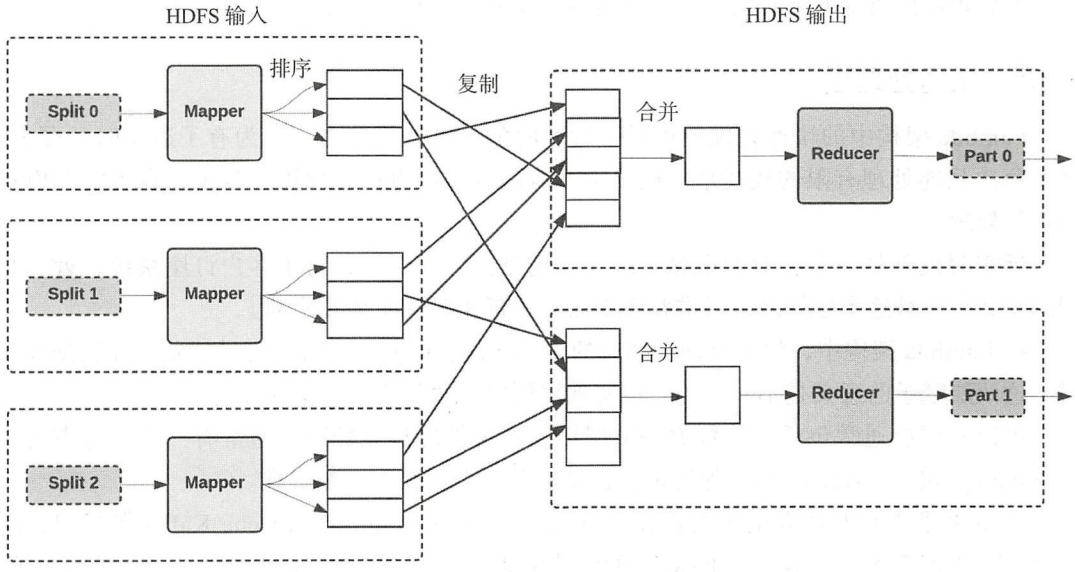


图 4-6　MapReduce 内部机制

在这里可以看到，当作业初始化时，会被划分为多个 Mapper 作业。Mapper 作业的数

非卖品，仅供非商业用途或交流学习使用

量通常取决于块（block）的大小和要处理的数据量。从作业流程的角度来看，使用者总是可以指定 Mapper 作业的最大数量，但是 Mapper 的数量总是受到指定的最大 Mapper 数的限制。当我们想要限制可以用于批处理作业的 CPU 核心数量时，这是非常有用的。

正如前面所说的那样，数据块大小在批处理中扮演了一个重要的角色，因为 Mapper 的计算单位依赖块大小，作业会被分散到多个 Mapper 中去执行，它们的输入为数据块。

从一个较高的层面来看，典型的批处理作业的执行遵循以下流程：

1）作业驱动程序执行，该程序设置了作业上下文，如 Mapper、Reducer、数据格式化类等。

2）作业执行时读取的数据切割为数据块输入给 Mapper。

3）Mapper 的输出经过排序（sorted）和混洗（shuffled），然后输出给 Reducer。

4）最后，Reducer 在 Mapper 产生的中间数据上执行 reduce 函数，并将输出结果存储到 HDFS 中，存储格式已在作业驱动程序中指定好了。

上面的执行步骤看起来简单明了，但实际的作业执行由很多个阶段组成。在这里，为了让读者对 Hadoop 批处理有基本的了解而不至于陷入各种细枝末节，对处理步骤做了必要的简化。

在后面的章节中，将会再次讨论该主题，届时会介绍更多批处理层的技术细节。

Lambda 架构对批处理层的总体期望是提供高质量的、经过处理的数据，这些数据可以与快速处理层的近实时处理关联，从而得到近实时的、相当可靠和一致的信息。

4.3.5 快速处理层

Lambda 架构中的快速处理层提供了近实时的事件处理能力。因为有了近实时的要求，因此每次只能处理有限的数据量。机器学习算法或复杂事件处理算法都可以在近实时模式中进行处理。

近实时处理是一个相对的概念，因人和场景而异。例如，对于客户订座系统，近实时指的是 2 ～ 3 秒这个数量级，对推荐引擎而言，近实时指的是几分钟。

在 Lambda 架构中，快速处理层捕获的事件或消息同时也被批处理层捕获，在数据处理时，这两层赋予的意义是不一样的，在实现用例时，两层的作用是互补的。

快速处理层通常包括对从数据获取层接收的事件的流式处理，通常有一个消息中间件（messaging middleware），用于保证消息传输以及与数据获取层的解耦。

早期快速数据处理方案中通常使用 Apache Storm、Flume、Apache Kafka 等框架，同时结合基于消费者（consumer-based）的流式处理。Flume 仍然是该领域的一个比较流行的选项，不过最近的 Spark Streaming 以及 Flink 框架由于一些优良特性（对并行处理部署和流水线处理提供良好支持，操作简便，可水平扩展）而备受用户青睐。这些框架的操作方式

非卖品，仅供非商业用途或交流学习使用

有着非常具体的差异，其中一些将在后面进行解释。

4.3.6　Flume 用于数据获取

在数据湖实现中，Flume 在数据获取方面扮演了一个非常重要的角色。图 4-7 展示了 Flume 是如何进行流式处理的。

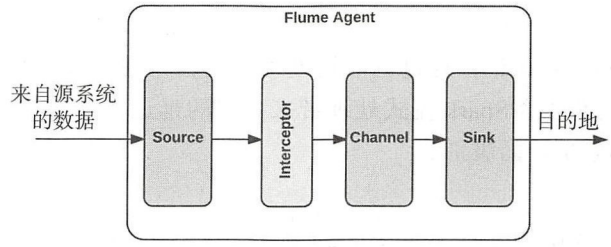


图 4-7　Flume 流式处理

这里显示的是 Flume 进程的一种较高抽象层次的架构图，由 Source、Interceptor、Channel、Sink 等组件构成。每个组件在处理事件 / 消息时负责特定的职能。

下面开始对 Flume 进行介绍。非常细节的知识请参考后面的章节，这里只需要掌握 Flume 组件的功能摘要。

1. Source 用于连接数据源

Source 组件可以从各种源系统捕获 Flume 事件 / 消息形式的数据。Flume 有大量内置的 connector，可连接到多种源系统，支持多种通信协议。

2. Interceptor 用于事件拦截

Interceptor 在事件 / 消息路由到 Channel 的过程中对它们进行转换。它们通常由事件序列化器（serializer）组成，在事件 / 消息抵达通道之前对其做必要的处理。

3. Channel 用于事件路由

Flume 同时支持持久化的 Channel 和基于内存的 Channel。Channel 为 Flume 事件 / 消息提供间接层，并支持多种模式将事件 / 消息从通道重播（replay）到 Sink。

4. Sink 作为事件目的地

从 Flume 的角度来看，Sink 代表的是目标系统的 connector。Flume 内置了很多 Sink connector 用于与企业中的多类系统连接，支持多种协议。它们的使用方式与 source 类似。

虽然 Flume 是早期用于近实时流式处理的方法，但它并不具备真正的可以适应事件 / 消息的近实时定制处理能力。Flume 的设置和部署在给定的时间点是静态的，具有确定的配置。若要改动，需要修改配置文件并重启 Flume。这对实现近实时用例增加了限制。

Flume 的这些缺陷很快被 Storm、Spark Streaming 之类的框架弥补了。在本书中，将 Lambda 架构应用在数据湖实践中时，考虑采用 Spark Streaming 和 Flink 框架。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

4.3.7 Spark Streaming

图 4-8 中显示的是一个极度简化后的 Spark Streaming 进程视图。Spark 最初被设计用于提升 Hadoop 批处理的速度，后来针对近实时处理，演化出 Spark Streaming。两种场景共用了一些基础构建块和模式。Spark Streaming 的主要构建块有 DStream、Receiver 和 RDD（Resilient Distributed Dataset，弹性分布式数据集）。Spark 最初用于优化批处理，之后用于近实时案例处理，两种处理方式非常类似。实际上，在近实时处理时，Spark Streaming 使用的是微批处理（micro-batch），通过调节批处理时间窗口就可以实现近实时处理。当然，批处理时间窗口的设置会给 Spark 流式处理带来一定的延迟，这会将近实时处理的响应速度限制在秒级，而不是更快的级别。

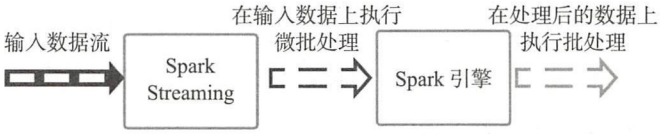


图 4-8 Spark Streaming

如图 4-8 所示，Spark Streaming 与具有实时数据流入的数据流协调工作，提供近实时处理能力。Spark Streaming 组件将传入的数据流划分为多个微批处理。然后将这些微批处理提交给核心的 Spark 引擎，该引擎对这些微批处理任务进行处理，批量产生处理后的数据。

1. DStream

对于输入或输出数据流而言，流的含义是一样的，它代表离散的 RDD 集合。Spark Streaming 提供了多种流，作为 Spark Streaming 框架的一部分，同时，多个技术框架也提供了对 Spark Streaming 的支持以及各自的 RDD 实现，可为 DStream 所用。

DStream 被提交给核心的 Spark 引擎处理之前，被划分为若干个微批处理，如图 4-9 所示。

图 4-9 Spark Streaming 中的数据流

2. DataFrame

DataFrame 代表数据处理操作被允许的时间窗口，被处理的数据集也是基于时间窗口的。此类操作中最常见的一种就是通过 Spark 会话在特定窗口数据集或 DataFrame 上

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

执行 SQL 查询。这允许对给定时间窗口内的数据进行快速分析和窗口操作（windowing operation）的执行。DataFrame 通常被视为列集合，可以被逻辑上看成是一个表，但是经过优化，用户可通过 Spark 框架提供的类 SQL 接口对其执行快速操作。

3. Checkpointing

对于需要提供 7×24 小时可靠服务的这类重要应用，Spark 必须能实现必要的容错能力，为此 Spark Streaming 同时支持针对元数据和数据的 Checkpointing。元数据 Checkpointing 包括配置、DStream 操作及用于恢复整个进程的批处理，而数据 Checkpointing 包括将正在处理的 RDD 持久化到可靠的存储介质上去。可为涉及数据转换的操作启用 Checkpointing。然而对于简单的处理，在可以容忍的情况下，某些故障级别可能不需要启用 Checkpointing。

4. Apache Flink

Apache Flink 作为一个技术框架，克服了 Spark Streaming 的限制，也对只执行一次（exactly once）处理提供了支持，具有很好的一致性。它逐行处理数据，并不受 Spark Streaming 微批处理那样的限制。它还支持基于时间的 windowing 功能，这些功能在执行事件关联时非常有用，同时保持处理流水线的灵活性和可扩展性。

Flink 的一些关键特性使之与众不同，并且非常适合用于迭代处理，这通常被归因于它的近实时处理能力。但是它也支持批处理。下面列举了 Flink 的一些关键特性：

- exactly once 处理，Flink 凭借该特性，成为一个可以信赖的能进行精确聚合计算的流式处理框架。Flume 通常不能胜任此类处理。Flink 提供了 Checkpointing 机制来实现容错。
- out of order 处理，Flink 支持处理乱序问题，它赋予 Streaming 层强大的能力，能按照预期的顺序对事件时间线（event timeline）进行处理。在一个典型的多线程环境中，事件很可能会乱序地流向下游系统。图 4-10 进一步展示了乱序问题。

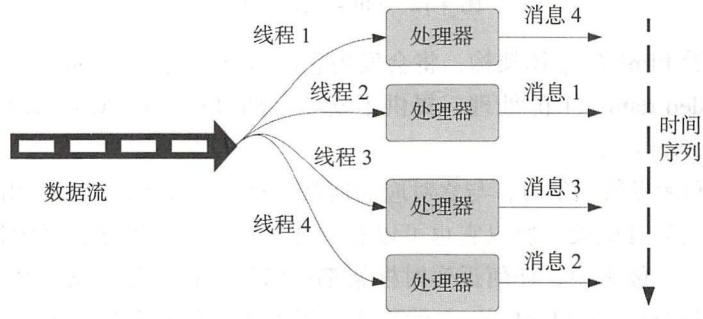


图 4-10　乱序场景

- 为流式事件提供开箱即用的 windowing 功能，不仅基于事件时间，还可基于计数和会话。当事件需要被分组在一起时，这项功能尤其有用。
- 故障恢复时不会损失数据，具有非常轻量级的进程状态管理，使得处理能从故障点

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

恢复，如图 4-11 所示。

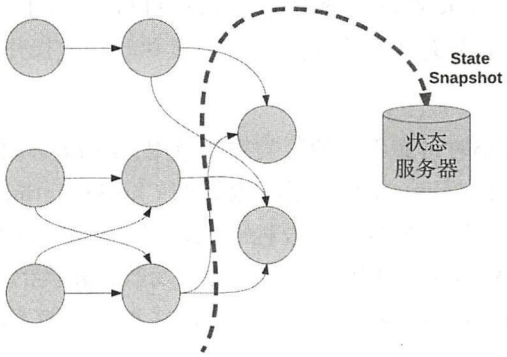


图 4-11 Flink 中的故障恢复

- Flink 被证明具备高吞吐和低延迟的特性。如前面所述，由于它不受微批处理约束的限制，与 Spark Streaming 相比传输延迟非常低，恰好是最合适的近实时事件处理框架。
- Flink 与 YARN 和 MESOS 作为资源管理器协同工作，并在可用节点上调度事件处理和故障恢复。

Flink 设计为运行在大型集群之上，实现时也对此进行了精巧的设计与优化。Flink 也支持 standalone 集群模式部署，此时能进行动态的流水线处理，参考图 4-12 中的范例。

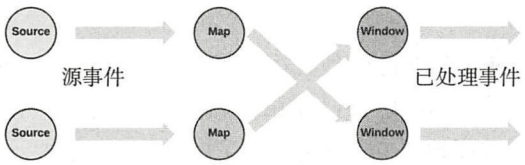


图 4-12 Flink 流式处理

如果仔细查看 Flink 的总体架构，将会发现它同时支持有界（bounded dataset）及无界数据集（unbounded dataset）的处理，提供了支持这两种模式的 API。总体架构如图 4-13（源自 flink.apache.org）所示。

当提到有界和无界数据集时，与之对应的通常是批处理和流式处理。由于 Flink 从根本上来看是一个流式处理框架，所以它也可以有效地执行批处理操作，因为批处理数据只是流式数据的子集。一般来说，任何近实时框架都可以用于批处理。反之并不成立，纯粹的批处理框架，例如 Hadoop MapReduce 就不能充当流式处理框架的角色，因为它是为批处理构建的，不能用于流式处理。即便将不同批次作业之间的时间间隔减到最小，也总会因为数据准备、处理和加载批处理的结果导致较大的延迟。

表 4-1 中列举了本章讨论过的 3 个框架的关键区别，这 3 个框架分别是 Flume、Spark Streaming 和 Flink。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

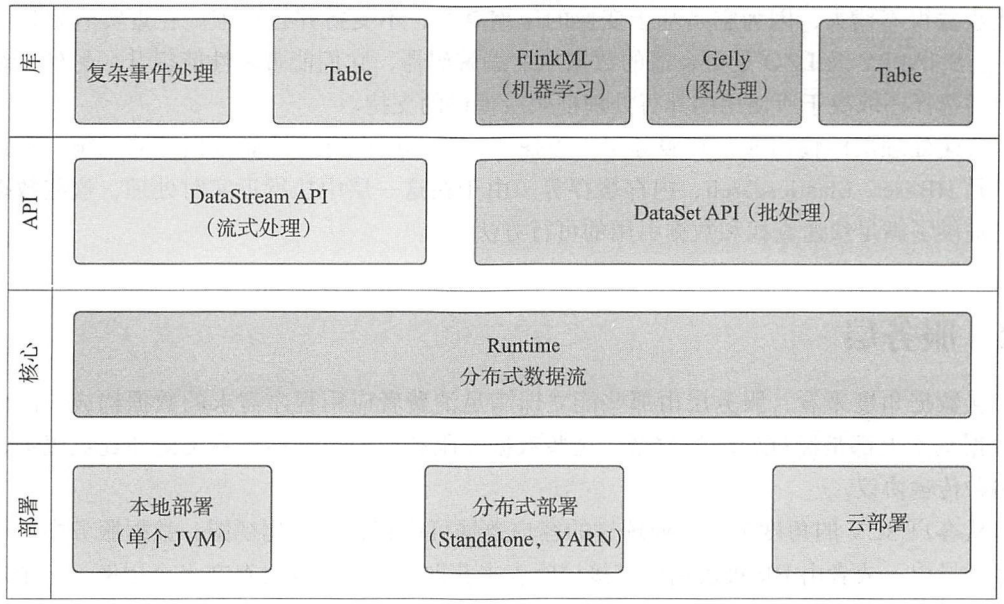


图 4-13　Apache Flink 架构

表 4-1　Flume、Spark Streaming 和 Flink 的关键区别

Flume	Spark Streaming	Flink
对于数据获取而言，Flume 主要充当事件生产者的角色。不过，通过实现自定义 Sink，Flume 可用于流式数据处理	提供流处理能力作为 Hadoop 框架的一部分，并提供跨集群节点拓扑的流式处理	可运行在集群之上的，基于内存计算的近实时处理框架
通过增加 Flume 配置中的 Sink 数量来提供可扩展性，配置是静态的，若更改，需要重启进程	支持对基于集群节点的拓扑的动态扩展	支持基于集群节点的处理拓扑的动态扩展
Flume 中数据处理以事件为单位，为实时场景提供了单次处理或批处理支持	仅以微批处理方式处理数据，因此会导致一定的延迟，对于某些关键的应用，可能达不到用户的预期	近实时方式处理事件
Flume 对每个事件至少处理一次（at least once），如果需要，会对事件进行重放，这会导致重复处理	支持 exactly once 处理，但是可能会导致数据丢失。同时也支持 at least once 处理，但是不会丢失数据	支持 exactly once 处理

除了并行处理外，复杂事件处理、自然语言处理（Natural Language Processing，NLP）和机器学习也在近实时处理中得到了非常有效的应用。这些算法中有一些适用于近实时处理，而很多不适合。处理中的任何延迟会影响整个处理时间，复杂系统的性能往往是由处理速度最慢的组件决定的。

数据压缩扮演了影响处理速度最关键的角色，它是影响吞吐率的最重要的因素之一。

从处理的角度来看，远程过程调用（Remote Procedure Call，RPC）中的数据压缩会带来时间开销，不过它确实为近实时处理减少了昂贵的 I/O 操作，从而带来可观的性能提升。对于这样的处理，选择正确的压缩编解码器（codec）是很重要的。例如，简单的 ZIP 压缩

仅供非商业用途或交流学习使用

性能收益得不偿失，因为它是基于变换的压缩算法，不支持并行处理。在近实时处理的背景中，SNAPPY 及 LZO 是更合适的数据压缩编解码器，它们能带来性能提升。另外，编解码器的选择还取决于所使用的并行处理框架所提供的支持。

快速处理层的输出通常在服务层中被捕获，服务层拥有高性能数据存储系统，常见的例子是 HBase、Elasticsearch、内存缓存等。由于在这一层中执行近实时处理，这些数据技术还提供了满足快速查找和数据引用的可行方法。

4.4 服务层

从数据角度来看，服务层由那些能立即满足消费者应用程序需求的数据构成。因此这些数据基本上都是经过处理的数据。这些数据经服务层的数据存储系统向外提供服务，支持多种传输协议。

从客户 SCV 的角度来看，处理过的客户数据可能存在于物化视图、数据服务中，或者经服务导出，或者由 BI/ 报表用例直接访问表来获取。可能会同时存在多个用例，每个用例中的数据表示及访问方式都可能有所不同。

典型的服务层架构如图 4-14 所示。

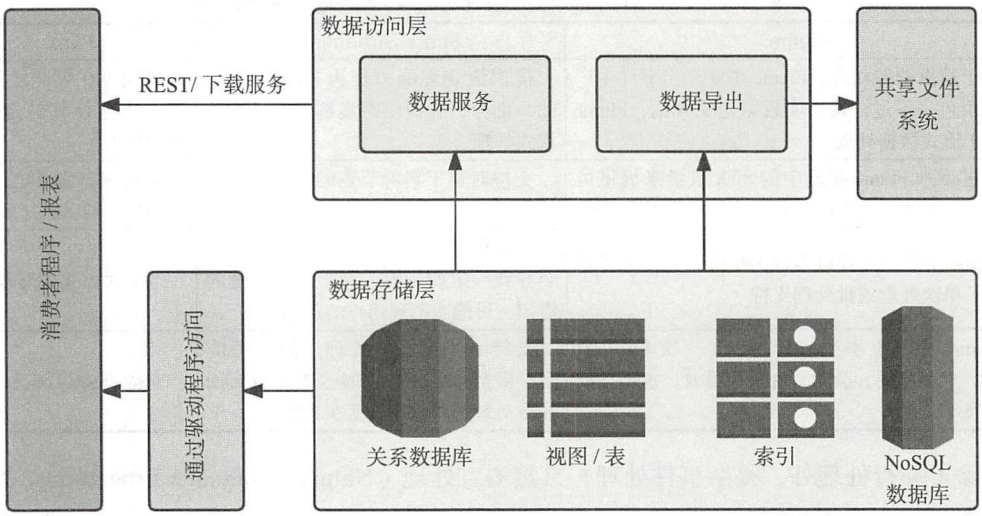


图 4-14 服务层

从这个表示中可以看出，一个服务层也可能有多个活动部件，大体来说，它们是数据存储层、数据访问层或数据传输层。

4.4.1 数据存储层

数据存储层可以由多种数据库组成。需要多种数据库的原因是需要使用不同的数据类

仅供非商业用途或交流学习使用

型来满足不同的需求，单一的数据库不能包揽一切。服务层背后的核心思想是它应该能够按需提供服务，因此多数据库在这里找到了用武之地。一般来说，服务层需要对批处理层和快速处理层的数据视图进行组合来对外提供服务。

1. 关系数据库

服务层中能发现很多实用工具，因为需要集成大量的报表及 BI 工具，这些工具通常工作于标准数据库（关系数据库）之上。这可能涉及从批处理层按预定的时间间隔将数据填充到数据存储系统中。

2. 大数据背景中的表与视图

某些应用需要直接集成表或视图中的数据，因此这些表或视图必须与服务层中的数据同步。一般情况下通过数据驱动程序来访问这些信息（表、视图中的数据），当使用 Hive 和 Impala 时，需要通过本地连接库来实现该目的。

3. 索引数据服务

索引数据用于快速搜索，通常数据访问层的数据服务需要用到索引数据。索引数据可按需以 REST/SOAP 端点（endpoint）的方式允许外部访问。索引层通常由基于 Lucene 的引擎来承担，该引擎同时也能提供快速查询功能，查询能感知被索引文档集近实时的变化。这些索引也可以选择性地服务于数据全集，并且对某些时间敏感型的应用也是极有帮助的。需要确保这些索引是为了支持高性能和可扩展性而构建的，因为它们处理了更多的实时服务负载。

最常见的构建数据服务的框架是 Spring Boot，紧随其后的是 Dropwizard。所有这些框架支持 JAX-RS 2.0 规范，并与 Swagger 等服务定义工具很好地结合，为构建和发布通用 REST 服务提供了优秀而全面的能力。

为了保持数据湖中数据的一致性，这些服务都应是只读的，因为它们的主要作用就是提供数据，而不应该公开可改变数据的端点，这是因为数据只能在前面介绍过的 Lambda 架构的数据处理环节中被修改。

4. NoSQL 数据库

此类数据库对那些想直接通过本地 NoSQL 驱动快速查找或读取数据的应用非常有用。NoSQL 数据的访问也可以被封装在数据服务之后，与索引数据的案例类似。在这里，数据需要以优化存储和支持预期访问模式的方式进行建模。它同时也应该支持高性能和可水平扩展的架构，就像索引数据一样，因为 NoSQL 也可以在近实时处理中扮演至关重要的角色。

4.4.2　数据访问层

任何应用程序的数据访问层都负责根据预期的访问模式从底层数据库访问数据。在这里，我们可以大致根据从服务层拉取还是推送数据来对数据访问层分类。

仅供非商业用途或交流学习使用

在较高的层次来看，数据推送指的是数据从服务层流出，其中，数据被服务层推到其他系统；而数据拉取指的是其他系统主动从服务层拉取，如果数据可用，服务层应支持预期的格式定义及传输协议。

1. 数据导出

数据可以通过预先设置好的 cron 任务从数据仓库中导出。许多应用程序倾向于根据需求定期执行数据转储。服务层必须提供所需的工具、进程和调度程序来支持数据导出。这样的数据导出作用显著；然而，为了获得最好的结果，需要确保尽可能多的流格式（txt、csv、可扩展标记语言、JSON 等）可用于数据导出，相对于其他格式（如 .doc、.pdf、.xlsx 等），这种方式更能保证性能的一致性。

2. 数据发布

服务层也可能需要对外发布处理后的数据。大多数来自近实时处理的输出，下游系统可能对种情况更感兴趣。服务层扮演的是事件中心的角色。大多数情况下，将此类事件暴露在一个 topic 上，这样多个消费者可以消费这些事件。然而，处理速度缓慢或消费者不可用可能导致在数据湖中堆积大量的消息。因此，该组件还应考虑出现故障的场景，确保平滑恢复，同时保持服务层的健康和功能。

如果要对基于推送或拉取来对数据服务进行分类，可参考表 4-2。

表 4-2 基于推送或拉取对数据服务分类

推送（push）	拉取（pull）
数据导出（data export） 数据发布（data publish）	关系数据库访问 大数据表 / 视图 通过数据服务访问 NoSQL 或索引

在本书中，表 4-3 所示的技术可以用于构建数据湖的数据服务层。

表 4-3 可构建数据湖的数据服务层的技术

数据服务层组件	技　术
关系数据库	PostgreSQL
大数据表 / 视图	Hive、Impala
索引	Elasticsearch
NoSQL	HBase、Couchbase
数据服务	Spring Boot Service
数据导出	Hadoop MapReduce、Sqoop、Pig Scripts
数据发布	JMS、Kafka

4.5 总结

在本章中，讨论了如何通过选择特定的框架和技术应用于 Lambda 架构。我们将利用

仅供非商业用途或交流学习使用

前面建议的这些框架和技术来实现第 1 章中的用例，本书后续章节中将逐步介绍如何搭建一个功能完备的数据湖。图 4-15 中简要展示了本章中选择的技术框架，这些技术与基于 Lambda 架构的数据湖中的各功能模块 / 层对应。

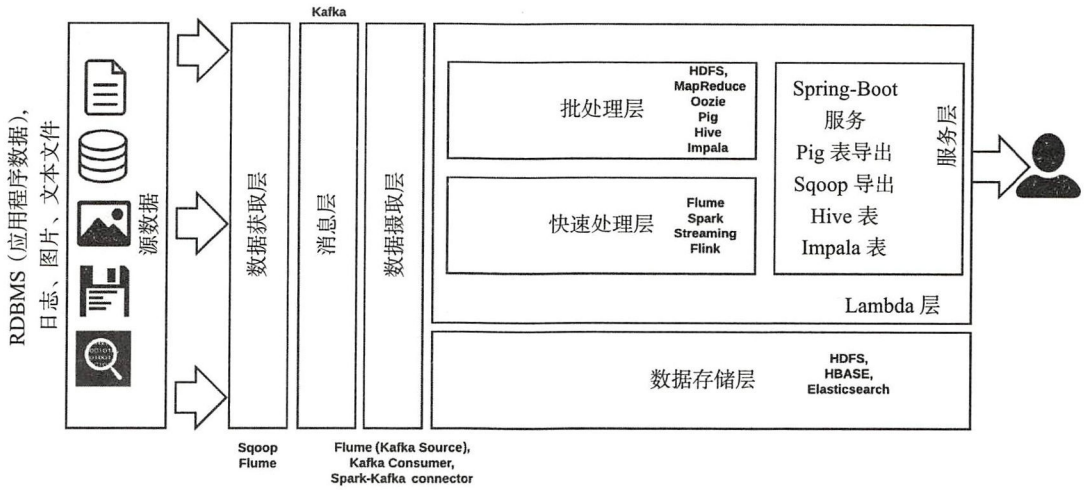


图 4-15　技术路线图

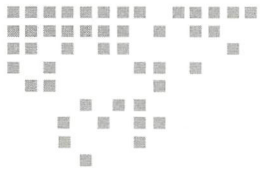
仅供非商业用途或交流学习使用

第二部分 Part 2

数据湖的技术组件

此部分向读者介绍许多技术，这些技术将是数据湖实现的一部分。此部分中的每一章都涉及一种技术框架，它们逐步被用于构建数据湖和用例，即单一客户视图（SCV）。几乎每一章中讨论的技术的所有重要技术细节都将以整体的方式被覆盖，因为过于深入的探讨超出了本书的范围。此部分共分 6 章，如下所示：

- 第 5 章　基于 Apache Sqoop 的批量数据获取
- 第 6 章　基于 Apache Flume 的流式数据获取
- 第 7 章　使用 Apache Kafka 构建消息层
- 第 8 章　使用 Apache Flink 处理数据
- 第 9 章　使用 Apache Hadoop 存储数据
- 第 10 章　使用 Elasticsearch 存储全文索引



Chapter 3 第 5 章

基于 Apache Sqoop 的批量数据获取

现在开始讨论基于 Lambda 架构的数据湖的一些必要元素，围绕数据湖构建，最迫切的是从各种源系统捕获数据，从源系统捕获数据的过程称为数据获取。

数据可以从多个系统中获取，这些系统中的数据以多种形式存在。每种数据格式都有对应的特殊处理方式，因此从各种源系统获取数据到数据湖中集中处理成为可能。

在本章中，将先探讨如何从关系数据源中获取数据，如关系数据库管理系统（Relational DataBase Management System，RDBMS），同时探讨了一些特殊的模式。当需要从关系数据源中获取数据时，Apache Sqoop 是一种被广泛使用的框架。Sqoop 也属于 Hadoop 生态系统，它在处理关系数据源时有很大的优势。

在本书的当前部分我们会提及多种技术（当然，本书的目标不是介绍所有的技术），下面将介绍那些真正用于实现数据湖的技术。

5.1 数据湖背景中的数据获取

从各种源系统捕获数据的过程被称为数据获取。在我们的数据湖中，有专门的功能模块 / 层用于处理数据获取（事实上，它是系统的第一层）。

Apache Sqoop 是数据湖中一种主要的数据导入工具。后续小节将对 Sqoop 做深入介绍，致力于帮助读者系统掌握数据获取层相关技术。

5.1.1 数据获取层

在第 2 章中，读者已经对数据获取层有初步的了解了。该层负责从各种各样的源系统

中获取数据，然后将它们导入数据湖中。读者可借助图 5-1 回忆数据湖的架构以及数据获取层的具体功能。

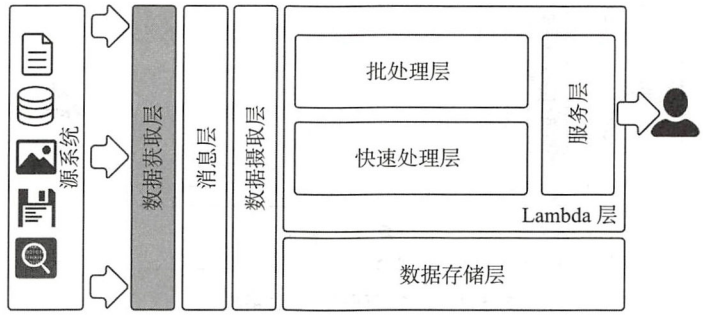


图 5-1　数据湖——数据获取层

数据获取层能处理以下事务：

- 批量数据：必须以批处理方式或者微批处理方式处理的数据，数据量与应用场景有关。Sqoop 能够处理各种应用程序存储在关系数据库中的海量数据。微批处理指的是操作更频繁但是数据量相对较小的数据操作。Sqoop 并不是微批处理的最佳选项，此时应采用 Apache Flume（详见第 6 章）。
- 高速数据：从微批处理到批处理，数据量可能会从几 MB 增加到几 TB。其中微批处理需要被数据获取层高效处理而不能有任何性能瓶颈。高速数据需要从两个方面来理解，其一是数据涌入数据获取层的速度（微批处理操作频繁但是随机，批处理通常在特定时段触发）；其二是涌入数据获取层的数据量。
- 多格式数据：即可处理多种文件格式（XML、JSON、TEXT 等），多种不同的结构化或非结构化数据。非关系数据——二进制数据；多源数据，如物联网传感器数据；服务器日志；机器生产日志；图像数据；视频数据等——这些都需要被高效处理。
- 结构化 / 非结构化数据：上面提到了结构化 / 非结构化数据，这里单列出来是因为它们的特殊意义。此外，半结构化数据也应提及，它介于结构化数据与半结构化数据之间。在第 1 章中，已经详细介绍过这几类数据了，在此不再赘述。
- 与多种技术或系统的集成：企业中有多种商业程序及 Web 应用，数据获取层能以比较便捷的方式与这些应用程序或数据存储集成，将数据导入数据湖的数据存储层中。

5.1.2　批量数据获取——技术路线图

为了支撑本书中的案例以及构建数据湖，我们在数据获取层使用了两种不同的技术，分别是 Apache Sqoop 与 Apache Flume。本章将深入介绍 Sqoop，第 6 章将会详细介绍 Flume。

图 5-2 所示是与数据湖概念架构对应的具体技术，这种架构表述方式将贯穿全书。后

面每章将会聚焦于一种技术，以及它们与整个数据湖架构的相关性，到本书的第三部分，将对数据湖架构做总体介绍。

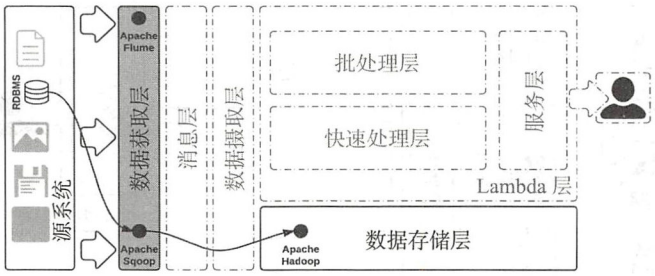


图 5-2 数据获取层技术路线图

为了与我们的案例保持一致，将用到一些传统的商业关系数据库。本书的案例中使用 PostgreSQL 存储用户数据。我们将分别连接到一个内网（如 B2B）应用程序及一个外网（如 B2C）应用程序，这些程序中都保存了用户的个性化信息。数据湖将这些分散的个性化数据汇集到一起，用于导出单一客户视图。

企业对企业（Business to Business，B2B）应用程序指的是企业间或企业多个部门间使用的程序。

企业对用户（Business to Consumer，B2C）应用程序指的是企业用于与用户交互的程序。

5.2 为什么使用 Apache Sqoop

Sqoop 是一个非常常用的用于向 Apache Hadoop 传输数据的工具。

在数据获取层，我们采用 Apache Sqoop 作为主要的技术框架。该层有很多可选的技术框架，这意味着这些框架之间往往可以互换。其他选项将会在 5.10 节进行详细介绍。

Apache Sqoop 是一种主流的，用于在关系数据库、传统数据仓库、NoSQL 数据库与 Hadoop 之间传输数据的工具。实践证明，Apache Hadoop 与传统数据库之间的数据传输非常麻烦，Sqoop 能以非常便捷的方式有效解决这个问题。

Sqoop 能以非常优雅的方式帮助用户在前面提及的存储系统之间传输数据，基于这个原因，Sqoop 被选用为数据获取层的技术框架。

Sqoop 也能让关系数据库、数据仓库、NoSQL 数据库与 Hadoop 生态系统轻松集成。Hadoop 生态系统包括但不限于 Apache Oozie、Apache HBase、Apache Hive 等工具。

Apache Oozie 是一个基于服务的用于管理 Hadoop 作业的工作流调度系统。

HBase 是一个开源的，分布式的非关系数据库，它由 Java 语言实现，借鉴了 Google BigTable

的设计理念。

Apache Hive 是一个数据仓库基础架构，它构建在 Hadoop 之上，用于处理数据统计、查询、分析等任务。

——维基百科

5.2.1　Sqoop 简史

Sqoop 最早由 Cloudera 开发和维护，后来在 2011 年 6 月 23 日孵化为一个 Apache 项目。在 2012 年 4 月，Sqoop 晋级为 Apache 顶级项目。从那时起，所有版本发布都由 Apache 委员会来控制。到本书完成之时，Sqoop 的稳定版本为 1.4.6，它于 2015 年 5 月 11 日发布。

由于 Sqoop 1（1.x.x 版本）的一些内在缺陷，驱使社区去改进它，这导致了 Sqoop 2 的诞生。不过在本书中，仍然采用 Sqoop 1 版本的例子，而不是 Sqoop 2。笔者会尝试尽量让读者了解更多细节，会对两个版本的 Sqoop 做详尽介绍，当然也包括何时应该采用哪个版本。

5.2.2　Sqoop 的优势

下面列举了 Apache Sqoop 的优点，这也是在数据获取层采用该框架的原因：

- 支持多种结构化数据的传输，如 Postgres、Oracle、Teradata 等。
- 由于数据被传输、存储到 Hadoop 中，因此 Sqoop 可以将 ETL 的某些环节剔除，转移到 Hadoop 中执行，因为在 Hadoop 执行这些操作代价更低，速度更快，处理更有效。
- Sqoop 可以以并行方式传输数据，因此它的执行速度可以很快，同时能降低成本。
- 帮助从主机（mainframe）中读取序列数据（sequential data），并集成至目标系统中。这里说的帮助不仅仅是限制对主机的使用，也包括削减执行某些代价高昂作业时消耗的主机硬件资源。
- Sqoop 可以将结构化数据存储系统中的数据传输至 Hadoop 中，而 Hadoop 中主要用于存储非结构化数据。数据传输过程允许用户组合使用两种类型的数据，用于满足各种分析目的，处理方式快速、高效并且能降低成本。
- 拥有一个扩展机制，通过它可以构建或拦截多种 connector。这可以用于定制现有的 connector，并且可以根据用例需求进行调整。有许多内置的 connector，例如 MySQL、PostgreSQL、Oracle 和其他知名数据库。由于它具有可扩展的能力，许多公司已经编写了支持良好的企业级 connector。例如，Oracle connector 是由 Quest Software 开发的，而 VoltDB connector 是由 VoltDB 自己开发的。
- 除了基于 JDBC 的 connector（适用于各种关系数据库系统），也有基于原生工具的 connector，后者性能更好。
- Sqoop 支持多种文件格式，如 Avro、Text、SequenceFile 等。

Avro 是 Apache 的 Hadoop 项目开发的远程过程调用和数据序列化框架。它使用 JSON 来定义数据类型和协议，并以紧凑的二进制格式序列化数据。

——维基百科

SequenceFile 是由二进制键 / 值对组成的 flat 文件。它在 MapReduce 中作为输入 / 输出格式被广泛使用。

—— Hadoop wiki

5.2.3 Sqoop 的劣势

尽管 Sqoop 拥有种种优点，但是它仍然有很多内在的缺陷，下面列举了一些 Sqoop 的主要缺点：

- 使用了 JDBC 与关系数据库连接，这种连接比较低效。
- 在进行分析时，Sqoop 会执行各种 Map Reduce 任务，这些任务非常耗时，因为会产生大量非规范化（denormalized fashion）的数据 join 操作。
- Sqoop 以批量方式传输数据，这样会给源系统带来很大的压力，当源系统所在服务器已经有较重负荷时，不适合使用 Sqoop 来传输数据。

5.3 Sqoop 的功能

就数据湖而言，用户需要从传统应用软件及数据存储系统中提取数据，而提取出来的数据量往往非常大，此时需要将这些数据灌入 Hadoop 中去。Sqoop 就是这么一种能轻松将传统企业数据导入 Hadoop 的工具：

SQL to Hadoop == SQOOP

图 5-3 展示了 Apache Sqoop 的基本功能。它提供了工具用于从 RDBMS 导出数据到 Hadoop 文件系统，以及从 Hadoop 文件系统导出数据写回到 RDBMS 中。

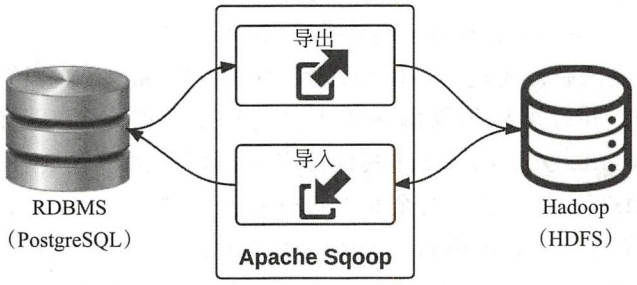


图 5-3 Sqoop 的基本功能

在我们的范例中，将存储在 RDBMS（PostgreSQL）中的数据导入 Hadoop 文件系统

非卖品，仅供非商业用途或交流学习使用

（HDFS）。本书并不会为 Sqoop 的数据导出费太多笔墨，但是会在本章中做简单介绍，以便让读者对 Sqoop 的功能有足够的了解。

在撰写本书时，Sqoop 已经有两个主要版本了，分别为 Sqoop 1 和 Sqoop 2。后面的内容中将会详细地介绍 Sqoop 1 和 Sqoop 2，并有对两者差异的粗略比较，读者很容易理解其差别。前文中已经介绍过了，书中案例使用 Sqoop1，这是因为 Sqoop 2 还在开发完善中，如果使用不稳定的 Sqoop 2，可能会带来各种问题。

下面借鉴了 Sqoop 官方文档中的一个图（见图 5-4），揭示了 Apache Sqoop 1 的基本架构。

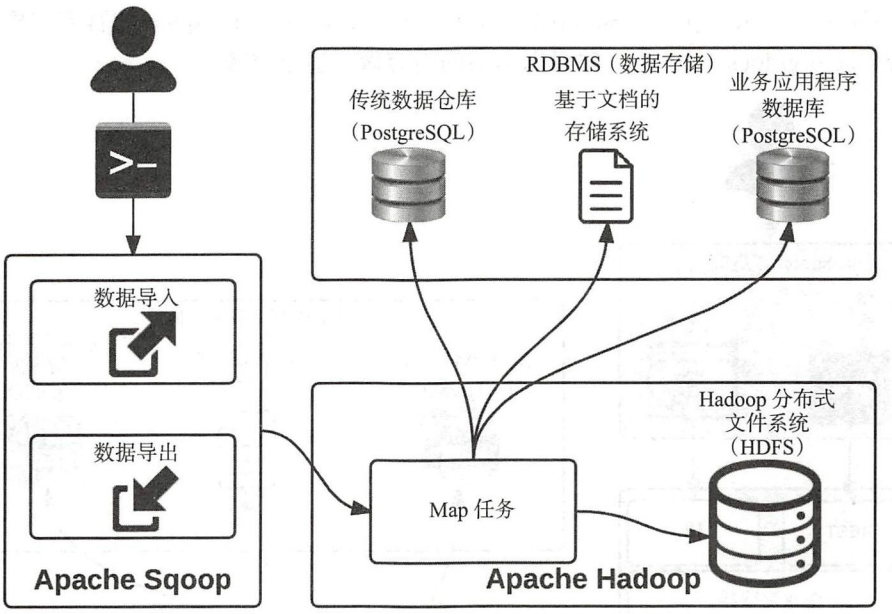


图 5-4　Sqoop 1 的概念架构图

Sqoop 的工作流程非常直接。用户通过命令行与 Sqoop 交互，命令行支持多种命令。输入命令后开始执行，它会触发 Hadoop 的 Map 任务，在 Map 任务中会连接到目标 RDBMS（这里使用了 JDBC），然后读取数据并存储到 Hadoop 文件系统中去。Sqoop 1 存在一个“痼疾”，它比较原始，使用了 JDBC 连接，而这种连接方式对各种用例来说过于笨拙低效。

JDBC 是一种 Java 编程语言中的应用程序编程接口（API），它定义了客户端是如何访问数据库的。它也是 Oracle 公司的 Java 标准版（Java Standard Edition）平台的一部分。

——维基百科

下面的内容将会带领读者简单浏览一下 Sqoop 2，从逻辑上来讲，它是 Sqoop 1 的升级。

非卖品，仅供非商业用途或交流学习使用

5.3.1 Sqoop 2 的架构

Sqoop 2 的工作方式与 Sqoop 1 非常相似，但是 Sqoop 2 引入了很多用户友好、易于使用的特性，相比之下，Sqoop 1 的使用较为低效。Sqoop 2 提供了一个基于 Web 浏览器的工具，同时也保留了客户端（在 Sqoop 1 中，客户端是唯一选项），用户只需要在某台机器上安装 Sqoop 一次，就能在多个地方访问它。Sqoop 2 同时也提供了大量的 RESTful API（更多细节请参考 Apache Sqoop 文档：https://sqoop.apache.org/docs/1.99.5/RESTAPI.html），这些文档能帮助用户在建设数据湖的各种场景中有效地集成 Sqoop。

图 5-5 展示了 Sqoop 2 的架构细节，读者明显能发觉其与 Sqoop 1 工作方式的不同。图中阴影部分标识了 Sqoop 2 与 Sqoop 1 的不同之处。该图参考了 Sqoop 的官方文档（https://sqoop.apache.org/docs/1.99.5/），并根据本书的内容做了适当修改。

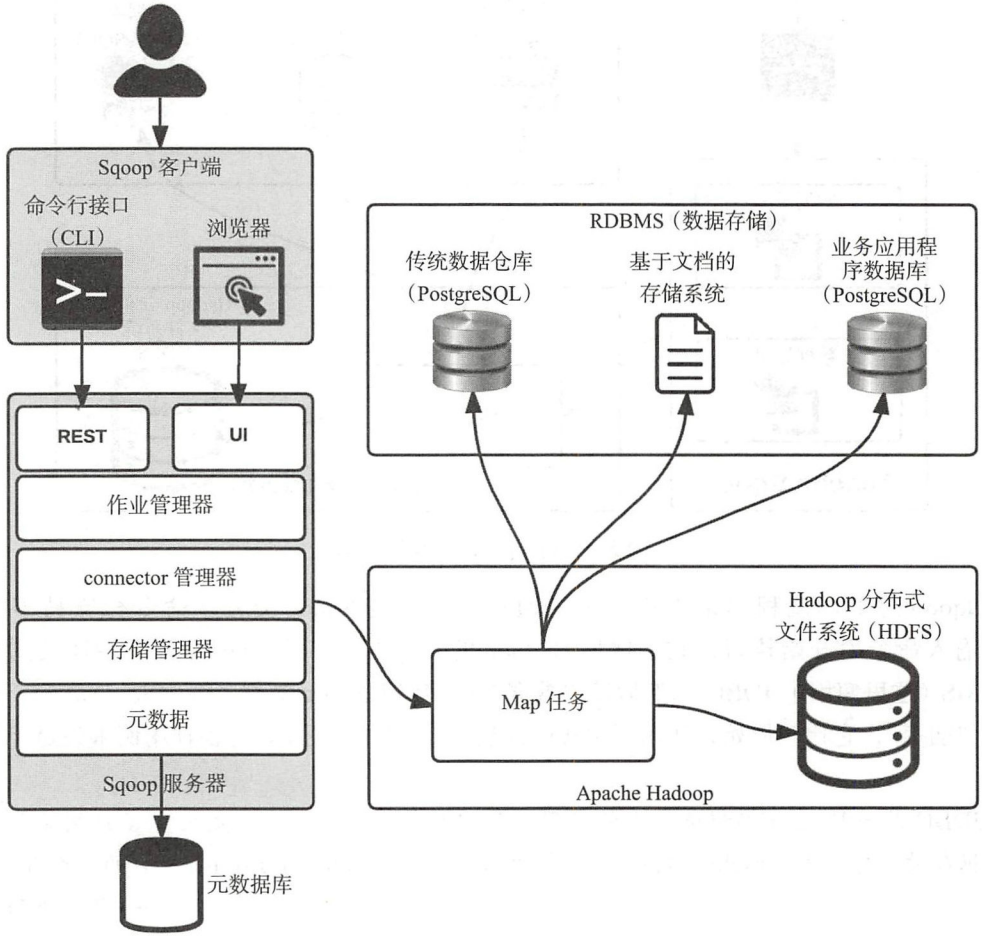


图 5-5 Sqoop 2 的概念架构图

如图 5-5 所示，阴影部分是 Sqoop 2 中新增的模块。Sqoop 2 中引入了一个服务模块，

非卖品，仅供非商业用途或交流学习使用

并且新增了浏览器形式的新客户端，用户可以使用浏览器接口与 Sqoop 交互，同时屏蔽了命令行的复杂性。由于服务模块的存在，用户可以在多台机器上与 Sqoop 交互，而 Sqoop 1 的客户端必须与其在同一台机器上。Sqoop 2 的服务模块里有很多组件，与之对应的是不同的新特性，这些特性在图 5-5 中有所体现。服务模块中有一个叫作元数据（metadata）的组件，用于存储描述数据的数据。元数据极大地提高了 Sqoop 的易用性，通过使用这些存储在数据库中的元数据，很多重复性的操作可以被消除。

5.3.2　Sqoop 1 与 Sqoop 2

理解 Sqoop 两个版本的区别很重要，本节致力于解释二者的差别。首先介绍 Sqoop 2 的主要设计思路，然后逐项对比与 Sqoop 1 的区别。

Aapche 上的 Sqoop 博客中曾深入介绍过这些话题，下面是新出现的并且仍在不断演进的 Sqoop 2 架构的主要设计思想。

- 易用性
- 易扩展性
- 安全性

下面将依照这些设计思想，逐项对比 Sqoop 1 与 Sqoop 2。

1. 易用性

Sqoop 2 与 Sqoop 1 相比，前者在客户端与 Hadoop 之间添加了一个功能模块，用于提高 Sqoop 的易用性。表 5-1 列举了两个版本的设计思路的差别。

表 5-1　Sqoop 1 与 Sqoop 2 在易用性方面的对比

Sqoop 1	Sqoop 2
客户端选项只有命令行	客户端选项包括命令行或浏览器（通过使用 Hue）接口
纯 Client 架构	Client-Server 架构
客户端只能在安装 Sqoop 的机器上运行	服务启动后，允许用户从多台机器访问 Sqoop
当考虑集成时，需要与其他工具（如 Apache Oozie）紧密绑定	对外暴露了 REST API，非常便于与其他系统集成
每次安装需要单独配置 connector 与驱动。每个客户端需要了解具体的配置信息，然后才能与数据存储系统连接，执行操作	由于服务模块的存在，所以只需要在一个地方配置 connector 与 JDBC 驱动
缺乏良好定义的基于角色的访问控制	由于服务模块作为中央访问入口，基于角色的访问和任务执行成为可能
由于需要用户阅读很多文档并手工配置各种选项，所以极易导致错误	基于浏览器访问，会提示用户输入所有的必选项，这样能有效减少错误

Hue（Hadoop User Experience）是一种开源的 Web 接口，它支持 Apache Hadoop 及其生态系统，遵循 Apache v2 license。

——维基百科

非卖品，仅供非商业用途或交流学习使用

很显然，Sqoop 2 的设计要比 Sqoop 1 先进很多，前者的很多优势都来自于一次性集中式的服务模块的安装。

2. 易扩展性

现在来了解一下 Sqoop 2 的第 2 个设计亮点。这种设计对现代企业非常关键，因为企业日常数据存储在各种各样的应用程序中。

如表 5-2 所示即为 Sqoop 1 与 Sqoop 2 在易扩展性方面的对比。

表 5-2 Sqoop 1 与 Sqoop 2 在易扩展性方面的对比

Sqoop 1	Sqoop 2
仅支持 JDBC 风格的 connector	除了 JDBC 以外，可以构建和使用其他类型的 connector。由于有这种特性，所以 Sqoop 2 非常灵活，通过维护对应的 connector，多种存储系统的数据写入变得非常轻松
通用的 connector 功能没有抽象出来，因此开发 connector 时非常复杂，开发者需要知道 Sqoop 的很多烦琐的底层细节	通用的 connector 功能被抽象出来了，connector 提供者只需要实现某些核心功能，这有利于在实现 connector 时屏蔽那些我们无须了解的复杂细节
编写 connector 的复杂度很高	复杂度大大降低，因为 Sqoop 的底层工作细节已经被抽象出来了，connector 提供者只需要实现数据传输的核心逻辑
没有强制要求用户选择、验证 connector 的有效性，而这些操作都能有效帮助消除错误。connector 选择是隐式的	显式的强制要求选择 connector 类型，这能有效排除因 connector 选择不当而引起的错误

3. 安全性

Sqoop 2 还有另外一个关键设计理念，在设计与开发之初就考虑了安全性。对于现代企业来说，最关注的是数据安全性，而 Sqoop 2 能帮助企业实现各种粒度的访问控制。

表 5-3 对比了 Sqoop 的两个版本在这个非常关键的方面的差异。

表 5-3 Sqoop 1 与 Sqoop 2 在安全性方面的对比

Sqoop 1	Sqoop 2
没有基于角色的访问控制	基于角色的访问控制（role-based access control），这受益于 Sqoop 的集中式的服务模块
仅提供 Hadoop 安全校验	提供了 Hadoop 安全校验与基于角色的安全校验
与外部系统集成时，只提供了有限的安全校验（用户名 / 密码）	安全校验方面，对外部系统也使用基于角色的访问控制
客户端直接访问 Hive 或 HBase	服务器只需要有访问权限，而客户端需要有访问权限及定义良好的角色
每个客户构建各自的连接（connection），这些连接无法被复用	先创建连接类的对象，然后把它们放入连接池，之后多次重复使用。管理员可以创建连接对象，普通用户依据各自的操作角色预设使用连接来执行各种任务
不具备资源管理策略	具备资源管理策略。这个特性可决定连接的使用方式以及每个角色如何执行任务（数据导入或导出）

很显然，Sqoop 2 在安全性这个关键方面比 Sqoop 1 更有优势。在现代企业中，安全问题至关重要，尤其是关于数据如何被访问，以及特定数据应该由组织内哪些人访问。

非卖品，仅供非商业用途或交流学习使用

4. Sqoop 1 与 Sqoop 2 的各自适用场景

在本书撰写时，Sqoop 2 仍在开发中，并没有完整实现其设计的所有特性，版本也尚未稳定。因此笔者决定在本书中使用 Sqoop 1 演示相关案例。然而，很显然 Sqoop 2 有着压倒性的优势（前文已经详细叙述过了），读者应该在合适的时机用 Sqoop 2 替代 Sqoop 1。

5.3.3　Sqoop 的功能

在本节中，让我们来进一步讨论 Sqoop 的实际功能。当在命令行输入一个命令，将会执行下面这些操作：首先会启动一个 Map 任务，在这里，会通过合适的 connector 连接到目标 RDBMS，然后检索相关的数据。在 Map 任务结束之后，会将这些数据交由 Reduce 任务进行处理，在这里被检索到的数据会被持久化到 HDFS/HBase/Hive 中。

5.3.4　使用 Sqoop 导入数据

当输入数据导入命令时，Sqoop 的数据导入工具会使用各种可用的 connector API 连接到 HDFS，将单个或多个表中的数据导入 HDFS 中。在导入数据时，RDBMS 中的每一行对应 HDFS 中的一条记录。根据数据类型的不同，记录可能存储为文本文件（text file）、文本数据（text data）或顺序文件（sequence file），当数据为二进制时，则为 Avro 文件。

图 5-6（参照 Apache Sqoop 博客中绘制的 Sqoop 数据导入示意图）详细描述了 Sqoop 数据导入工具的功能，这里是从 PostgreSQL 导入数据至 HDFS。

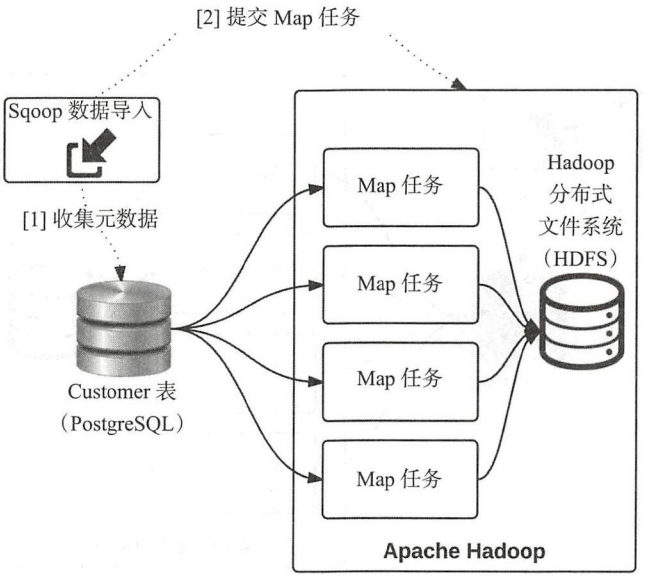


图 5-6　Sqoop 的数据导入

在 Sqoop 执行真正的数据导入之前，它会分析数据库，构建相关的元数据。构建好的

非卖品，仅供非商业用途或交流学习使用

元数据用于导入数据库表格中所需的数据，有些情况下可能会导入整个表格。

Sqoop 也提供了多种选项，基于这些选项进行数据导入。从数据表导入的数据存储为单个或多个 HDFS 文件（根据数据源的大小）。导入格式为 CSV（逗号分隔），数据表中的每一条记录对应 CSV 文件中的一行。Sqoop 也可以设置选项将数据导入为指定文件格式（如 Avro 或文本文件）。

在本章后续部分将向读者展示如何使用 Sqoop 命令行从 PostgreSQL 向 HDFS 中导入数据。本节只涉及 Sqoop 的数据导入功能及其底层原理。

5.3.5 使用 Sqoop 导出数据

Sqoop 的数据导出机制与数据导入类似。当导出命令被执行，Sqoop 收集数据导出所需的元数据，然后根据数据量及数据特性生成合适数量的 Map 任务，之后执行真正的数据导出操作，将数据导出到 RDBMS 中。它通过可用的 connector 最终将数据持久化写入数据库中。有些 connector 有一个临时表（staging table）的概念，其中数据首先被收集（分阶段），然后实际移动到目标数据库。临时表策略有个好处，当数据导出遇到故障时，因为临时表中的数据是可用的，整个工作不需要重新执行，相反的是它很可能被重复使用。

数据导出与数据导入非常类似，图 5-7（参照 Apache Sqoop 博客）揭示了 Sqoop 数据导出的内部工作机制。

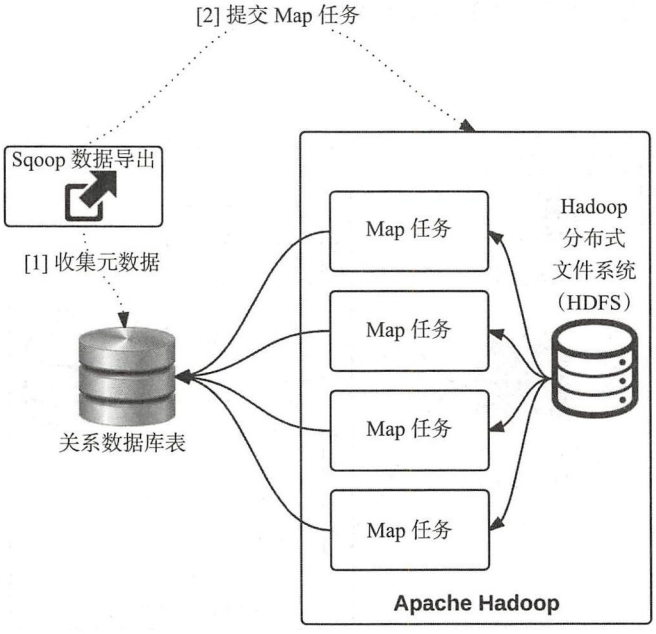


图 5-7 Sqoop 数据导出工作机制

尽管我们的案例并不要求展示 Sqoop 的数据导出功能，不过在下一节中还是会介绍

非卖品，仅供非商业用途或交流学习使用

Sqoop 数据导出相关的重要命令以便让读者对 Sqoop 有个完整的了解。

5.4　Sqoop connector

Sqoop connector 在 Sqoop 任务中起到如下作用：

- 连接到目标数据库（import 或 export）。
- 从数据库中抽取数据（import）。
- 加载数据到数据库中（export）。

Apache Sqoop 允许对自己插入代码的形式进行扩展，用于特定数据库系统的数据传输。该功能是 Sqoop 的可扩展框架的一部分，Sqoop 的所有版本都支持该机制。Sqoop 1 中已经实现了可扩展框架，Sqoop 2 在此之上进一步扩展，添加了很多新特性（本章 Sqoop 1、Sqoop 2 比较的部分已经涉及该特性）。Sqoop 2 对定义良好的 connector API 做了更好的集成。

当 Sqoop 进行数据传输时，两个组件开始起作用，它们分别是：

- 驱动（driver）：JDBC 是 Sqoop 连接到数据库的主要机制之一。在 Sqoop 中，驱动指的是 JDBC 驱动程序。JDBC 是 Java 开发工具包（JDK）规定的规范，它由各种抽象类和接口组成。任何 RDBMS 都应提供符合 JDBC 规范的驱动程序供客户端程序连接至它们。这些驱动程序是专有的，并且经常有与之相关的 lisence，在此基础上可以使用它们。为了让 Sqoop 正常工作，这些驱动程序可能需要由用户自己来安装，然后才能正常使用 Sqoop。由于这些驱动程序是由数据库系统供应商实现，它们会尽最大努力去提高驱动的性能和效率。
- 连接器（connector）：用于运行 Sqoop 任务（Job），它要求传输所需数据的元数据。connector 有助于检索这些元数据，并帮助以最有效的方式传输数据（import 和 export）。JDBC 是主要的连接机制之一，它使用 SQL 语言进行数据提取和加载，但是每个数据库系统都有一些被称为“方言”（dialect）的技巧。connector 使用这些方言高效地传输数据。Sqoop 集成了默认的 JDBC connector（通用的），能与 JDBC 及 SQL 兼容的数据库系统协同工作；但由于其通用的性质，它可能不是传输数据的最佳方式。还有其他内置（built-in）connector 和外部（external）专用 connector，将在下面的小节中详细讨论。

图 5-8 揭示了 Sqoop 客户是如何使用这些组件来连接到目标数据库并传输数据的。

在 Sqoop 1 中，当执行一个命令时，Sqoop 首先分析命令行参数，扫描 Sqoop 已安装的最合适（更有效、更高的执行性能）的 connector。它同时扫描内置的和手动安装的连接，从中选择最佳的 connector。如果它不能找到合适的 connector，作为最后的手段，它将使用内置的也是通用的 JDBC connector。一旦它选择了一个 connector，它就会寻找最合适的驱动程序，而且大部分时候都有一个与特定数据库和连接配套的驱动程序。但是，对于 JDBC

非卖品，仅供非商业用途或交流学习使用

驱动程序，必须使用命令行参数显式地提供驱动程序。

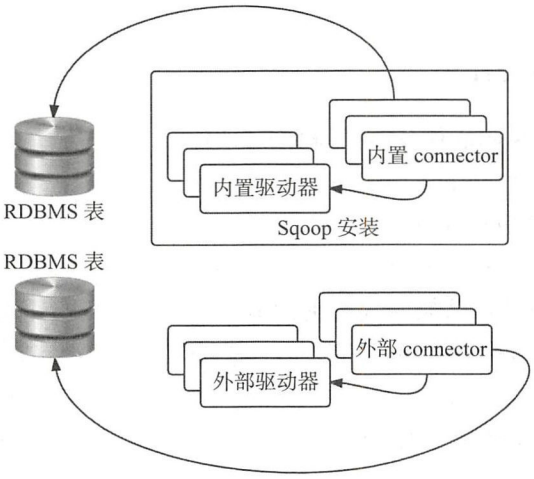


图 5-8 Sqoop connector 组件及其工作原理

Sqoop 2 与 Sqoop 1 有一个较大的区别，在 Sqoop 2 中，connector 必须显式指定，而在 Sqoop 1 中是隐式指定的。

Sqoop connector 类型

Sqoop connector 可大致分为以下几类：

- 内置的 connector（built-in connector）：即 Sqoop 安装包中默认集成的 connector。内置 connector 又可以进一步分成以下 3 类。
  - 通用的 JDBC connector（generic JDBC connector）：在 JDBC 及 SQL 范畴内，此类 connector 可以连接到任意的数据库系统。当 Sqoop 找不到其他可用的 connector（默认或手动安装）时，JDBC connector 是最后的手段。
  - 特殊的 connector（specialized connector）：此类 connector 包括 Sqoop 安装包默认集成的常见数据库系统的 connector。Oracle、MySQL，PostgreSQL 连接均属于此类。
  - 快速 connector（fast-path connector）：使用数据库特殊工具用于提高数据传输性能的内置 connector 都属于此类连接。MySQL 与 PostgreSQL 都提供了类似的基于原生工具的 connector，原生工具天然具有高性能、高吞吐等优点。
- 可插拔的外部 connector（pluggable external connector）：默认的内置 connector 之外，Sqoop 允许嵌入外部 connector 处理与特殊的数据源相关的事务，以期提供更好的数据传输能力。此类 connector 被称为外部 connector。外部 connector 都是特殊 connector，主要由数据库供应商提供和维护。由供应商自己管理，这些 connector 通常是高性能的，使用不同的数据库的原生工具做数据迁移工作。处理不同的 NoSQL 数据库 connector 都是现成的，如 Couchbase connector。

5.5　Sqoop 对 HDFS 的支持

Sqoop 为 HDFS 数据导入导出而生。显然，它在架构上能支持数据在源系统和目标系统之间的导入和导出。事实上，如果我们观察 Import 和 Export 这两个关键字，自然而然会想到这个工具是完全关于向 HDFS 导入数据以及从 HDFS 向外导出数据的。Sqoop 也支持增量式的数据导出和导入，当然，需要一个额外的属性 / 字段来追踪数据库的变化情况。

Sqoop 也支持多种文件格式以应对存储优化问题，支持的文件格式包括 Apache Avro、orc、parquet 等。Avro 与 parquet 是常见的 HDFS 中的文件存储格式，而 orc 文件格式能提供更好的性能和数据压缩支持。由于 Avro 与 parquet 格式文件更易于维护，HDFS 增强了相关支持，以及支持多值字段（multi-value field）和搜索模式，因此它们更受欢迎。

Avro 是 Apache Hadoop 项目开发的远程过程调用和数据序列化框架。它使用 JSON 定义数据类型和协议，并以紧凑的二进制格式对数据进行序列化。它主要用于 Apache Hadoop，既可以提供持久数据的串行化格式，也可以提供用于 Hadoop 节点之间或客户端程序到 Hadoop 服务之间通信的有线格式（wire format）。

——维基百科

5.6　Sqoop 运行范例

我们将使用谷歌云平台（Google Cloud Platform）运行本书出现过的例子。在本书中，截屏和代码将会出现，这样读者在本书的最后部分会有一个功能齐全的数据湖，它可以与企业中实际存在的数据库连接起来。

由于本书是全书首次介绍具体技术框架的章节，必然会涉及软件安装和代码，本章将安装一些软件、工具或库，在后面的章节中将会详细介绍它们。在 Sqoop 背景中，一些安装和命令是不需要的，但是在云中运行书中例子则需要这些安装或命令，因为云平台中的节点是非常干净的节点，没有安装任何东西。

本书提供了大量可用的例子，并且已经在 CentOS 7 中测试过了，CentOS 7 是本书示例的标准运行平台。

5.6.1　安装与配置

本书将基于以下基本约定和假设来讨论各种安装问题：

1）假设用户为非 root 用户，但是 sudo 用户配置为无须密码登录。

2）所有下载均假设下载到同一个目录下，该目录可用环境变量 ${DOWNLOAD_DIR} 来引用。具体目录可以通过以下命令配置到相应的环境变量中，同时该环境变量也必须添加到 ~/.bashrc 文件中去：

```
export DOWNLOAD_DIR=<path of download directory>
```

3）除 RPM 包以外，所有软件包都假设在用户目录中安装，因为 RPM 包安装可能涉及系统级更改、软件包安装，这里需要用到 sudo 账户。

现在我们来处理一些棘手的问题，可能会涉及一些代码。本节详细介绍了每一个细节，便于读者阅读和理解相关知识。

1. Java 安装与确认

1）为了确保能正常运行本书中的范例，建议仅安装 Oracle JDK。

2）请从 Oracle 官网下载最新版本的 Java JDK。对于 Linux 操作系统，建议下载 RPM 包。在编写本书时，最新版本为 jdk8（参考 http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html），不过 JDK 版本变化较快，读者在实践时最好查询一下最新的 JDK 版本。

3）为了从链接地址下载安装包，请在 CentOS shell 中运行下面的命令。这里使用的链接指向 download.oracle.com，不过读者需要注意，下载链接是动态变化的，每次下载读者需自己确认并复制可用的下载链接。

```
wget --header "Cookie: oraclelicense=accept-securebackup-
cookie"
http://download.oracle.com/otn-pub/java/jdk/8u131-b11/d54c1d3a0
95b4ff2b6607d096fa80163/jdk-8u131-linux-x64.rpm
```

4）一旦安装包下载成功，可执行下面的命令进行完全安装。

5）可通过下面的命令安装已下载的 RPM 文件：

```
rpm --install <JDK_RPM_FILE>
```

6）设置 JAVA_HOME 环境变量，指向 JDK 安装目录，通常被安装在 /usr/java 子目录中。

```
export JAVA_HOME=/usr/java/default
```

7）配置 PATH 环境变量：

```
export PATH=$PATH:$JAVA_HOME/bin
```

8）将上面这条语句添加到 ~/.bashrc 文件中去。

9）通过下面的命令显示 PATH 环境变量值来验证 Java 的安装目录：

```
echo $PATH
```

可通过 java -version 命令查看已安装的 JDK 版本。

2. Hadoop 安装与确认

1）前面提到过，我们将使用开源软件 Apache Hadoop 演示本书中的例子，对应的版本

为 2.7.3。

2）使用 wget 命令下载链接对应的 Hadoop 版本至 ${DOWNLOAD_DIR} 中：

```
wget
http://www-eu.apache.org/dist/hadoop/common/hadoop-2.7.3/hadoop-2.7
.3.tar.gz
```

3）一旦下载完成，在 shell 中切换至任意用户目录，不妨使用环境变量 ${HADOOP_HOME} 来表示它（Hadoop 安装目录），通过下面的命令解压安装文件：

```
tar -xzvf ${DOWNLOAD_DIR}/ hadoop-2.7.3.tar.gz
```

4）同时，在 shell 中使用下面这些命令禁用防火墙，然后重启 VM：

```
systemctl stop firewalld
systemctl disable firewalld
```

5）为了便于演示范例，将 Hadoop 设置为伪分布式模式（pseudo distributed mode）。读者需要注意，在生产环境中，需要将系统设置为分布式模式（distributed mode），在第 9 章中，会进一步介绍 Hadoop 的安装配置细节。切换至 ${HADOOP_HOME} 对应的目录，更改相应配置。可参考 https://goo.gl/x6TC9q。

6）为 Hadoop 配置伪分布式模式时，如果出现 HOST KEY VERIFICATION FAILURE 类型错误，可在 shell 中执行下列命令：

```
ssh-keyscan -H -t rsa localhost  >> ~/.ssh/known_hosts
ssh-keyscan -H -t rsa 0.0.0.0  >> ~/.ssh/known_hosts
ssh-keyscan -H -t rsa 127.0.0.1  >> ~/.ssh/known_hosts
```

7）一旦 DFS 设置并运行，说明安装已经完成，可访问 http://localhost:50070 来确认安装是否成功。

8）对于满足演示本书中各种范例的需求而言，伪分布式安装是合理且足够的。

9）到目前为止，我们只运行了 DFS，而在后面数章中，将会涉及与 Hadoop 的其他服务，如 YARN、Hive、Hbase 等。演示 Sqoop 范例只需要用到 DFS。

10）使用下面的命令设置环境变量 HADOOP_HOME，令其指向 Hadoop 安装目录，同时配置 PATH 环境变量，并将它们追加到 ~/.bashrc 文件中去：

```
export HADOOP_HOME=<Hadoop-Installation-Directory>
export PATH=$PATH:$JAVA_HOME/bin:$HADOOP_HOME/sbin:$HADOOP_HOME/bin
```

3. Hue 安装与确认

Hue 提供了友好的用户界面用于访问各种 Hadoop 服务、服务调度、工作流，甚至可以浏览 HDFS 详情。Hue 的安装非常简单明了，甚至可以通过编译源码来安装。可执行下面这些命令来安装 Hue：

1）使用下面的命令来下载 Hue 3.1.1（.tgz 文件）：

```
wget
https://dl.dropboxusercontent.com/u/730827/hue/releases/3.11.0/hue-
3.11.0.tgz
```

2）使用下面的命令将 .tgz 文件中的内容解压到用户目录中：

```
tar -zxvf ${DOWNLOAD_DIR}/hue-3.11.0.tgz
```

3）Hue 的安装依赖 Python 2.7 以及其他软件包，这要求先安装操作系统相关的 package installer。为了演示本书中的范例，必须尝试在 CentOS 7 中安装所依赖的这些软件包，如下所列：

```
sudo yum install libffi-devel
sudo yum install gmp-devel
sudo yum install python-devel mysql-devel
sudo yum install ant gcc gcc-c++ rsync krb5-devel mysql openssl-
devel cyrus-sasl-devel cyrus-sasl-gssapi sqlite-devel openldap-
devel python-simplejson
sudo yum install libtidy libxml2-devel libxslt-devel
sudo yum install python-devel python-simplejson python-
setuptools sudo yum install maven
```

4）读者可参考 Hue 的 README.txt 文件，其中有安装说明，在 Hue 源码目录中执行下面的命令（该目录为 .tgz 文件中提取出来的内容）：

```
PREFIX=~/ make install
```

5）这里的 PREFIX 可认为是安装目录，不过它也可以是任意的用户目录，Hue 在安装时，会在 PREFIX 目录下创建一个 hue 子目录。

6）一旦安装完成，使用正确的 HDFS URL 配置 ${PREFIX}/hue/desktop/conf/hue.ini 文件，如下所示：

```
fs_defaultfs=hdfs://localhost:9000
```

7）使用下面的命令设置 ${HUE_HOME} 环境变量，并将它追加到 ~/.bashrc 文件中去：

```
export HUE_HOME=~/hue
```

（假设该目录在前面的安装步骤中已经创建好了。）

8）对 ${HADOOP_HOME}/etc/hadoop/hdfs-site.xml 文件中的下列属性进行配置：

```
<property>
  <name>dfs.webhdfs.enabled</name>
  <value>true</value>
</property>
<property>
  <name>hadoop.proxyuser.hue.hosts</name>
  <value>*</value>
</property>
<property>
  <name>hadoop.proxyuser.hue.groups</name>
```

```
  <value>*</value>
</property>
```

9）一旦 Hue 安装设置好，切换至 ${HUE_HOME} 目录，使用下面的命令启动 Hue 服务：

```
${HUE_HOME}/build/env/bin/supervisor
```

10）该服务应该以 daemon 进程方式运行，这样的话，当客户终端连接断开时，进程仍然在后台继续运行，可使用下面的命令来进行设置：

```
${HUE_HOME}/build/env/bin/supervisor -d
```

11）访问 http://localhost:8888，创建 Hue 用户账号，登录以后，就可以在 Hue 中查看 HDFS 文件视图了（位于右上角）。

登录 Hue 控制台之后，可能会报告出现了某些错误或者配置问题，不过不用担心，这些是可以忽略的，因为在后面将具体框架集成至 Hue 时，可逐个修正与之相关的配置。

4. Sqoop 安装与确认

随着 Hadoop 版本的不断发布，Sqoop 的功能与特性也随之急剧变化。正如前面提到过的那样，到现在为止 Sqoop 框架有 Sqoop 1 及 Sqoop 2 两个主版本。对于 Hadoop 领域而言，Sqoop 是一个较老的 ETL 框架，已经不再添加新功能 / 特性了。而 Sqoop 2 更新颖，设计理念更先进，它提供了 REST 风格的接口，不过还在不断完善中。本书特别地介绍了 Sqoop 的安装。

可通过二进制形式安装，软件下载、解压、安装参考下列步骤：

1）通过下面的命令下载与 Hadoop 2 兼容的二进制 Sqoop 安装包：

```
wget
https://www-eu.apache.org/dist/sqoop/1.4.6/sqoop-1.4.6.bin__hadoop-
2.0.4-alpha.tar.gz
```

2）将安装包中内容解压至用户目录，这里不妨解压到 ${SQOOP_HOME} 所指的目录，解压命令如下：

```
tar -zxvf <DOWNLOAD_LOCATION>/sqoop-1.4.6.bin__hadoop-2.0.4-
alpha.tar.gz
```

3）使用下面的命令配置 ${SQOOP_HOME} 环境变量，并将它们追加至 ~/.bashrc 文件中去：

```
export SQOOP_HOME=${SQOOP_HOME}
export PATH=$PATH:$SQOOP_HOME/bin
```

4）在 CentOS shell 中执行 sqoop --help 命令来验证 Sqoop 是否安装成功。

另外，也可以通过源码安装 Sqoop，参照下面的操作步骤：

1）下载最新版本的 Sqoop 源码：

```
wget http://www-eu.apache.org/dist/sqoop/1.4.6/sqoop-1.4.6.tar.gz
```

2）通过下面这行命令将压缩包解压到用户目录中，这里不妨用 ${SQOOP_HOME} 来引用该目录：

```
tar -zxvf <DOWNLOAD_LOCATION>/sqoop-1.4.6.tar.gz
```

3）遵照 COMPILING.txt 文件中的指令编译源码，或者使用 ssh 切换至 ${SQOOP_HOME} 目录中，执行下面这行命令：

```
ant release
```

4）在安装期间，如果报告有任何可执行文件缺失，那么请使用 package installer（yum install <required package>）来安装。在 CenoOS 7 中，可能遇到 AsciiDoc、LSB 及 xmlto 等可执行文件的缺失。

5）安装完毕后，将 Sqoop 源码文件夹下的 bin 子文件夹中的内容复制到 build/bin 目录中，并且将 build 目录复制到另外一个单独的文件夹中。

6）使用下面的命令配置 ${SQOOP_HOME} 环境变量，并将它们追加至 ~/.bashrc 文件中去：

```
export SQOOP_HOME=${SQOOP_HOME}
export PATH=$PATH:$SQOOP_HOME/bin
```

7）在 CentOS shell 中执行 sqoop –help 命令来验证 Sqoop 是否安装成功。

5. PostgreSQL（关系数据库）安装与确认

关系数据库方面，我们选择了 PostgreSQL，它既可以用来存储元数据，也可以用来存储关系数据。本章中的例子是关于 Sqoop 的，因此会涉及关系数据库，PostgreSQL 安装好以后，在后续章节中可以被重复使用。

PostgreSQL 几乎为所有主流操作系统提供了安装包。Linux 下的 PostgreSQL 安装包是标准安装包，可以通过原生的 package installer 来安装。在 CentOS 7 中，可按照下列步骤安装 PostgreSQL。

1）在终端中运行下面这条命令：

```
sudo yum install postgresql-server*
```

2）一旦数据库安装完毕，运行下面这些命令对数据库进行初始化：

```
sudo service postgresql initdb
sudo chkconfig postgresql on
```

3）启动 postgre 服务：

```
sudo service postgresql start
```

4）到现在为止，已经为 postgre 数据库系统创建了一个用户（非交互式），可执行下面这些命令对该用户的密码进行更改：

```
sudo -u postgres psql postgres
```

上面这行命令让系统用户进入 postgre shell，提示符为 postgres=#，执行下面的命令更改用户密码：

```
postgres=# password postgres
```

5）输入命令 q 可退出 postgre shell：

```
postgres=# q
```

6）下一步，需要修改新安装的数据库的数据目录中的配置。执行下面的命令来查看数据目录及 PostgreSQL 服务端口。默认端口为 5432。作为参考，执行下面这行命令，可以看到与执行 /usr/bin/postgres -D/var/lib/pgsql/data -p 5432 类似的输出结果：

```
sudo service postgresql status
```

7）设置 pgsql 的权限，以便后面能够修改其中的数据文件，可使用下面这行命令：

```
sudo chmod 777 -R /var/lib/pgsql
```

8）现在切换到 postgre 的数据目录，修改 pg_hba.conf（/var/lib/pgsql/data/pg_hba.conf）配置以允许本地或远程用户通过 postgre 用户凭据连接至数据库。下面是典型的 pg_hba.conf 配置文件：

```
# TYPE  DATABASE        USER            ADDRESS
                METHOD
# "local" is for Unix domain socket connections only
local   all             all
                                    md5
# IPv4 local connections:
host    all             all             127.0.0.1/32
          md5
# IPv6 local connections:
host    all             all             ::1/128
               md5
# remote connections
host    all             all             0.0.0.0/0
             md5
# Allow replication connections from localhost, by a user with
the
# replication privilege.
#local   replication     postgres
                              peer
```

```
#host    replication     postgres        127.0.0.1/32
           ident
#host    replication     postgres        ::1/128
                 ident
```

9）加粗的代码是修改后的配置，此时允许 username/password 方式登录。

10）现在对 /var/lib/pgsql/data/postgresql.conf 文件做如图 5-9 所示的修改。

```
# - Connection Settings -

listen_addresses = '*'          # what IP address(es) to listen on;
                                # comma-separated list of addresses;
                                # defaults to 'localhost'; use '*' for all
                                # (change requires restart)
```

图 5-9 修改 postgresql.conf 文件

11）重新对 /var/lib/pgsql 目录进行权限设置：

```
sudo chmod 0700 -R /var/lib/pgsql
```

12）使用下面的命令启动 / 重启 PostgreSQL 服务：

```
sudo service postgresql restart
```

此时，数据库已经安装配置完毕。为了能够远程访问数据库，必须确保以下事项：

- 数据库端口已开放并且可访问。
- 安装 Postgre 数据库客户端以便于用户执行查询，这里我们选择 pgAdmin4 作为 PostgreSQL 客户端。

6. HBase（NoSQL）安装与确认

1）从下面的 URL 处下载 HBase 0.98.24 的二进制安装包：

```
wget
https://archive.apache.org/dist/hbase/0.98.24/hbase-0.98.24-hadoop2
-bin.tar.gz
```

在本书撰写之时，HBase 的最新稳定版本为 1.2.4，但是笔者仍然选择了较旧的 0.98.x 版本的 HBase，因为考虑到较新的版本对 Sqoop 1 存在兼容性问题。

2）将压缩包内容解压到期望的目录，该目录不妨用环境变量 ${HBASE_HOME} 值来表示，参考下面的命令：

```
tar -zxvf downloads/hbase-0.98.24-hadoop2-bin.tar.gz
```

3）使用下列命令设置环境变量 HBASE_HOME 使之指向解压目录，并将它们追加到 ~/.bashrc 文件中去：

```
export HBASE_HOME=${HBASE_HOME}
export PATH=$PATH:$HBASE_HOME/bin
```

4）文件解压完毕，可以发现 HBase 配置文件位于 ${HBASE_HOME}/conf 目录中。

5）查看 <HBASE_HOME>/conf 目录中的 hbase-site.xml 文件，修改以下内容（centos 是配置范例中的用户名，可以用 VM 用户名替代；类似的 localhost 也可以用 IP 地址来替代）。

```
<configuration>
  <property>
    <name>hbase.rootdir</name>
    <value>hdfs://localhost:9000/user/centos/hbase</value>
  </property>
  <property>
    <name>hbase.zookeeper.property.dataDir</name>
    <value>/home/testuser/zookeeper</value>
  </property>
</configuration>
```

6）执行 <HABSE_HOME>/bin/start-hbase.sh 启动 HBase 服务，可通过 http://localhost:60010 来监控 HBase 服务的运行状况，如图 5-10 所示。

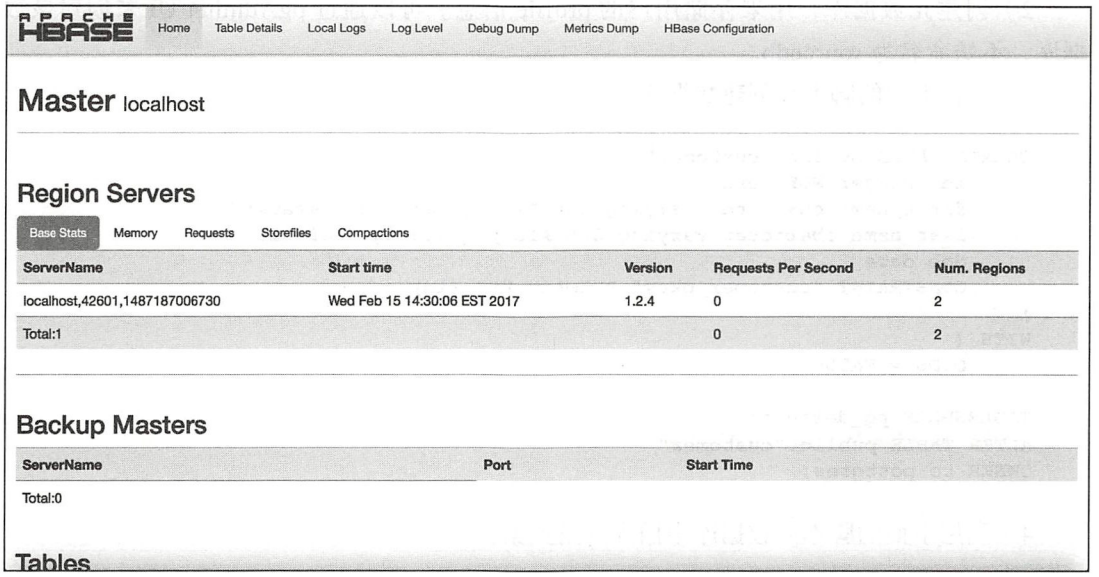


图 5-10　HBase 服务控制台

7）可通过启动 HBase shell，执行某些 HBase 查询来确定 HBase 是否安装成功。

① 通过执行 <HBASE_HOME>/bin/hbase 命令启动 HBase shell。

② 在 shell 中执行下面的查询来创建一个 test 表：

```
create 'test', 'cf'
```

③ 如果上面的表格创建成功，会提示基本设置已成功，然后会刷新 HBase 服务主页列表，新创建的表格将会出现在列表中，如图 5-11 所示。

Tables

User Tables | System Tables | Snapshots

1 table(s) in set. [Details]

Namespace	Table Name	Online Regions	Offline Regions	Failed Regions	Split Regions	Other Regions	Description
default	test	1	0	0	0	0	'test', {NAME => 'cf'}

图 5-11 HBase 安装验证

8）读者也可浏览 HBase 网站的 Get Started 页面了解更多参考信息。

以上是 HBase 的基本设置，不过对于演示 Sqoop 示例已经足够了。

5.6.2 数据源配置

本节将介绍数据源配置，经配置后，数据源中的数据被摄取到数据湖中。

1）用户（连同必要的用户凭据）通过客户端（pgAdmin4）登录 PostgreSQL 数据库。

2）创建元数据库，用来存储用户的 profile 信息。可以通过 pgAdmin4 UI 来创建该数据库，不妨命名为 sourcedb。

3）运行下面的脚本来创建数据库：

```
CREATE TABLE public."customer"(
     id integer NOT NULL,
     first_name character varying COLLATE pg_catalog."default",
     last_name character varying COLLATE pg_catalog."default",
     dob date,
     CONSTRAINT "customer_pkey" PRIMARY KEY (id)
)
WITH (
     OIDS = FALSE
)
TABLESPACE pg_default;
ALTER TABLE public."customer"
OWNER to postgres;
```

4）利用下面的脚本向数据库中插入少量数据：

```
INSERT  INTO public."customer"
values(0,'tomcy','john','1985-10-20');
INSERT  INTO public."customer"
values(1,'rahul','dev','1989-08-15');
INSERT  INTO public."customer"
values(2,'pankaj','misra','1982-08-10');
INSERT  INTO public."customer" values(3,'devi','lal','1990-05-06');
INSERT  INTO public."customer" values(4,'john','doe','1992-06-25');
```

经过以上步骤的设置，我们已经有源数据了，这些数据可以导入 HDFS，PostgreSQL 中的数据表如图 5-12 所示。

非卖品，仅供非商业用途或交流学习使用

id [PK] inte...	first_name character...	last_name character...	dob date
0	tomcy	john	1985-10-20
1	rahul	dev	1989-08-15
2	pankaj	misra	1982-08-10
3	devi	lal	1990-05-06
4	john	doe	1992-06-25

图 5-12　加载到 RDBMS 中的数据

5.6.3　Sqoop 配置（数据库驱动）

为 Sqoop 配置目标数据库驱动，请参考以下步骤：

1）因为我们安装的数据库为 PostgreSQL，请在 VM（CentOS）中执行下面的命令，从下面的 URL（本书撰写时的最新版本）所在位置下载 PostgreSQL JDBC 驱动：

```
wget
https://jdbc.postgresql.org/download/postgresql-9.4.1212.jre6.jar
```

2）JDBC 驱动放置到 ${SQOOP_HOME}/lib 目录下。

5.6.4　将 HDFS 配置为目的地

Sqoop 默认的数据传输目的地为 HDFS 和 RDBMS，只不过它们分别是导入与导出的目的地。不过 Sqoop 2 支持更多类型连接，如 Kafka、Kite 等。

Kite 是针对 Hadoop 的高级数据组件。它是一个 API 和一组能提升开发加速的工具。用户可以配置 Kite 如何将数据存储在 Hadoop 中，而不是自己构建和维护该基础架构。

——http://kitesdk.org/docs/current/

在 Sqoop 1 中，数据源与数据目的地都可以通过命令行参数来配置，在本章的多个范例中可以看到这种场景。

5.6.5　Sqoop 数据导入

Sqoop 提供了很多 shell 脚本，这些脚本中调用了 Sqoop job。所有的 Sqoop 操作都可以在单个 shell 脚本中执行，脚本可以在 ${SQOOP_HOME}/bin 目录中找到，如 sqoop.sh 等。

为了从配置好的数据源（如 postgresql）中导入数据到 HDFS 中，需要运行类似下面的命令：

```
bin/sqoop import --connect
jdbc:postgresql://<DB_SERVER_ADDRESS>/<DB_NAME>?schema=<SCHEMA> --table
<TABLE_NAME> --m 1 --username <DB_USER_NAME> --password <DB_PASSWORD>
```

非卖品，仅供非商业用途或交流学习使用


```
Where:
   <DB_SERVER_ADDRESS> → 数据库地址（主机名/IP）
   <DB_NAME>           → 数据库名称
   <SCHEMA>            → 数据库 Schema，如果源数据表存在
   <TABLE_NAME>        → 要导入的数据表表名
   <DB_USER_NAME>      → 访问数据表的 postgresql 用户名
   <DB_PASSWORD>       → 数据库用户的密码
   --me                → Map 任务数
```

在我们的例子中，数据导入命令如下所示：

```
sqoop import --connect jdbc:postgresql://<CentOS_IP>/sourcedb?schema=public
--table customer --m 1 --username postgres --password <password>
```

一旦数据导入完成，导入的数据可以通过 Hue 或 Namenode UI（Utilities/Browse the file system）来查看。

图 5-13 所示是 Hue 中的 HDFS 视图。

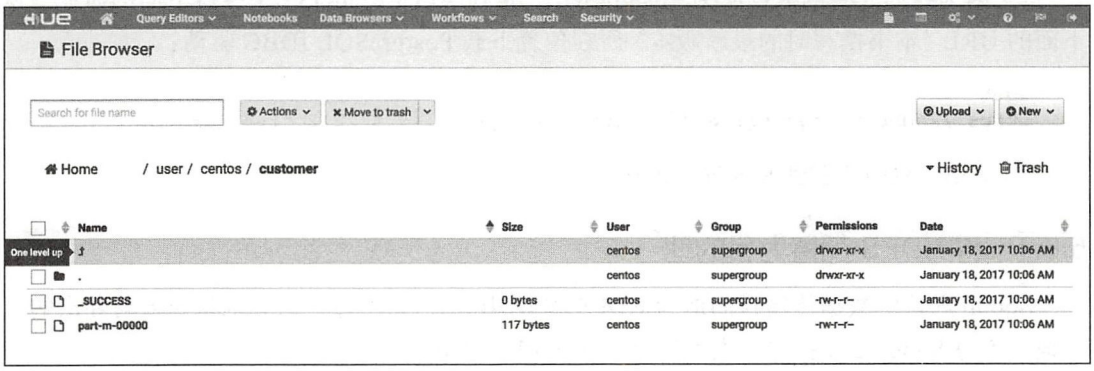


图 5-13 Hue 中的 HDFS 视图（HDFS 浏览器位于右上角）

图 5-14 所示是 Namenode UI 中的 HDFS 视图，内容与图 5-13 相同。

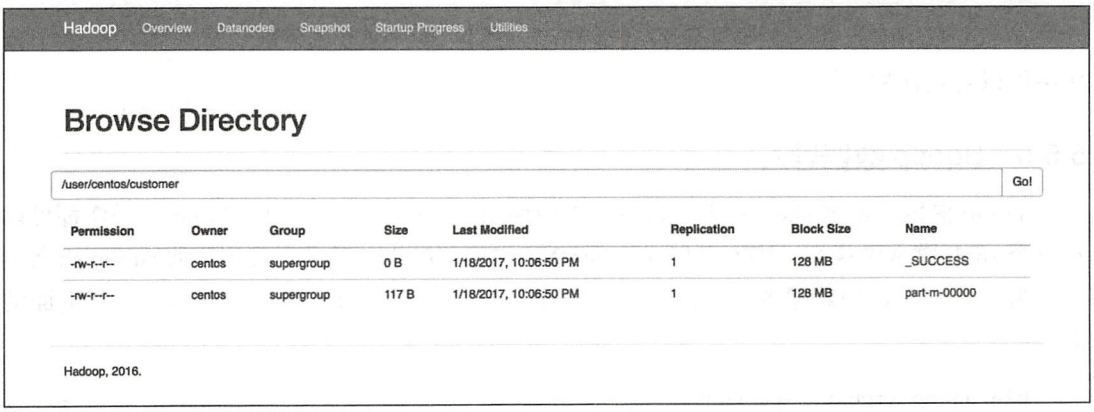


图 5-14 Namenode UI 中的 HDFS 视图（Utilities/Browse 下查看文件系统）

非卖品，仅供非商业用途或交流学习使用

在 Hue 中单击数据文件，可以查看文件内容，读者不难确定，该文件的内容正是最开始我们插入 PostgreSQL 的 sourcedb 数据库中的数据，如图 5-15 所示。

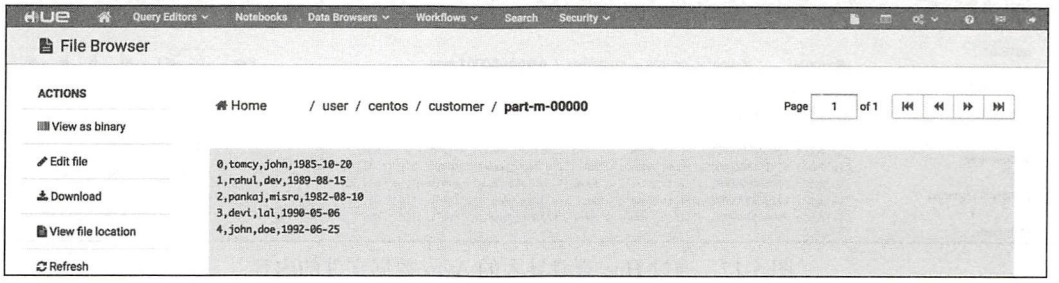


图 5-15　通过 Hue 查看 HDFS 文件内容

数据库中的数据在导入 HDFS 时被存储为文本文件格式，文本格式为默认格式，除非是经过特殊设置。Sqoop 也可以将导入 HDFS 的数据存储为 Avro 形式的二进制文件，仅需要对导入参数做少量修改，如下所示。

```
bin/sqoop import --connect
jdbc:postgresql://<DB_SERVER_ADDRESS>/<DB_NAME>?schema=<SCHEMA> --table
<TABLE_NAME> --m 1 --username <DB_USER_NAME> --password <DB_PASSWORD> --as-
avrodatafile --append
```

两次数据导入的差别可参考图 5-16，Hue 的 HDFS 文件浏览器显示了最近导入 HDFS 中的 Avro 文件。

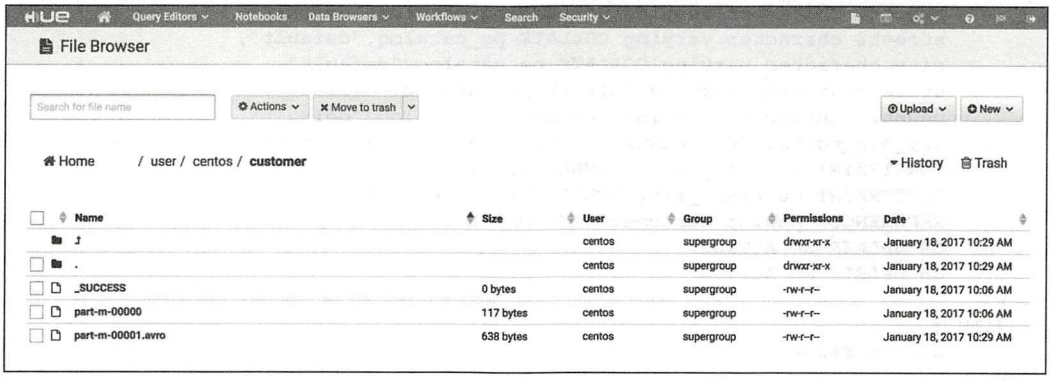


图 5-16　通过 Hue 浏览导入的 Avro 数据文件

在图 5-16 所示的页面中，单击 Avro 文件，将会再显示文件的内容，如图 5-17 所示。

从图 5-17 可以发现，Avro 文件的表示形式与文本文件略有不同，它是经过压缩的，因此能节省存储空间。

1. 导入整个数据库

Sqoop 也拥有将关系数据库中所有数据表导入 HDFS 的能力。典型用法如下所示：

非卖品，仅供非商业用途或交流学习使用


```
sqoop import-all-tables (generic-args) (import-args)
```

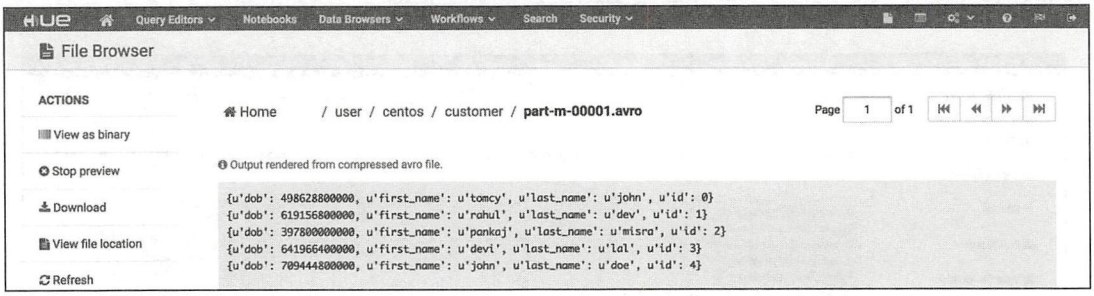


图 5-17 通过 Hue 查看导入的 Avro 数据文件的内容

与前面介绍的 import 命令相比，唯一的区别是 Sqoop 命令的 import-all-tables 选项。下面是对应的例子：

```
bin/sqoop import-all-tables --connect
jdbc:postgresql://<DB_SERVER_ADDRESS>/<DB_NAME>?schema=<SCHEMA> --m 1 --
username <DB_USER_NAME> --password <DB_PASSWORD> --as-avrodatafile
```

此时读者不难理解，为了演示该命令的执行，关系数据库中需要有多个数据表。因此，笔者建议再创建一个与 customer 相关的 address 表：

```
CREATE TABLE public.address
(
    id integer NOT NULL,
    custumer_id integer NOT NULL,
    street1 character varying COLLATE pg_catalog."default",
    street2 character varying COLLATE pg_catalog."default",
    city character varying COLLATE pg_catalog."default",
    state character varying COLLATE pg_catalog."default",
    country character varying COLLATE pg_catalog."default",
    zip_pin_postal_code character varying COLLATE pg_catalog."default",
    CONSTRAINT address_pkey PRIMARY KEY (id),
    CONSTRAINT customer_fkey FOREIGN KEY (custumer_id)
    REFERENCES public.customer (id) MATCH SIMPLE
    ON UPDATE NO ACTION
    ON DELETE NO ACTION
)
WITH (
    OIDS = FALSE
)
TABLESPACE pg_default;

ALTER TABLE public.address
    OWNER to postgres;
```

一旦数据表创建完毕，可使用下面的脚本导入范例数据：

```
INSERT INTO address (id, custumer_id, street1, street2, city, state,
country, zip_pin_postal_code)
VALUES (0, 0, 'd-40', 'chavez street', 'trivandrum', 'kerala', 'india',
'778908');
```

非卖品，仅供非商业用途或交流学习使用


```
INSERT INTO address (id, custumer_id, street1, street2, city, state,
country, zip_pin_postal_code)
VALUES (1, 1, '1-90', 'cooper street', 'mumbai', 'maharashtra', 'india',
'400056');
INSERT INTO address (id, custumer_id, street1, street2, city, state,
country, zip_pin_postal_code)
VALUES (2, 2, 'a-47', 'sector-11', 'noida', 'uttar pradesh', 'india',
'201311');
INSERT INTO address (id, custumer_id, street1, street2, city, state,
country, zip_pin_postal_code)
VALUES (3, 3, 'r-98', 'sector-37', 'gurgaon', 'haryana', 'india',
'122021');
INSERT INTO address (id, custumer_id, street1, street2, city, state,
country, zip_pin_postal_code)
VALUES (4, 4, '201', 'high street', 'austin', 'texas', 'us', '41101');
```

如果读者想查看 import-all-tables 的执行结果，则需要移除 HDFS 中已有文件，否则命令执行时会报错，报错信息类似下面所示：

```
Output directoryhdfs://localhost:9000/user/<username>/customer already exists.
```

为了移除 HDFS 中的内容，可在 shell 中执行下面的命令：

```
hdfs dfs -rm -r /user/<username>/*
```

当数据库导入完毕，可以看到所有数据表以级联方式遍历，以便于浏览所有信息，如图 5-18 所示。

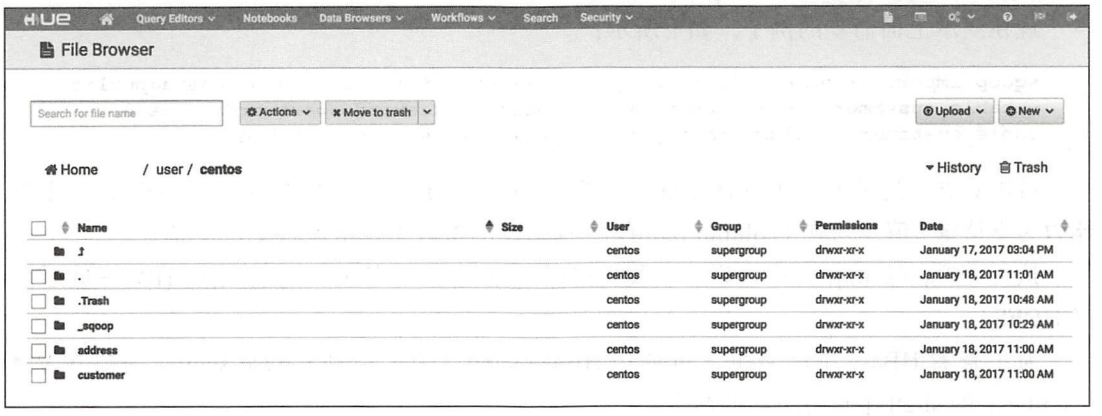


图 5-18　将数据库中所有表导入 HDFS

2. 导入部分表格

对于导入数据库中的部分表的情况，在 Sqoop 中并没有直接的方法，不过可以通过脚本来实现该目标。Sqoop 支持获取数据库中数据表的列表，然后通过循环筛选出想要导入的那些表，然后逐个将它们导入到 HDFS 中去。

非卖品，仅供非商业用途或交流学习使用

3. 从表格中导入选择的列

Sqoop 也提供了一个命令行选项用于导入一个表的若干列到 HDFS 中去。有两种方法可实现该目标，使用 --column 选项或使用 Sqoop 中格式自由的查询能力。两种方法的例子如下所示。

- 使用 --column 选项：

```
bin/sqoop import --connect
jdbc:postgresql://<DB_SERVER_ADDRESS>/<DB_NAME>?schema=<SCHEMA> --
table customer --columns id,first_name,last_name --m 1 --username
<DB_USER_NAME> --password <DB_PASSWORD> --as-avrodatafile --append
```

- 格式自由的查询能力：

```
sqoop import  --query 'SELECT c.*, a.* FROM customer c JOIN address
a on (c.id == a.id) WHERE $CONDITIONS'  -m 1 --target-dir
/user/foo/joinresults
```

4. 导入数据至 HBase

前面的安装步骤介绍中已经暗示过了，Sqoop 可以向 HBase 导入数据或者从 HBase 中导出数据。下面介绍如何使用 Sqoop 初始化从 RDBMS 向 HBase 导入数据。

```
bin/sqoop import --connect
jdbc:postgresql://<DB_SERVER_ADDRESS>/<DB_NAME>?schema=<SCHEMA> --table
<TABLE_NAME> --m 1 --username <DB_USER_NAME> --password <DB_PASSWORD>  -
hbase-table <HBASE_TABLE_NAME> --column-family <HBASE_COLUMN_FAMILY_NAME> -
hbase-create-table
```

现在演示上面命令的例子，如下所示：

```
sqoop import --connect jdbc:postgresql://<CentOS_IP>/sourcedb?schema=public
--table customer --m 1 --username postgres --password <password> -hbase-
table customer --column-family h_cust_col1 -hbase-create-table
```

可以看到，这里有一些与 HBase 相关的新参数。HBase 是一种 NoSQL 数据库，其主要能力为支持键 / 值及列族（column family），这里用的参数主要是表名及列族定义。

执行完 Sqoop import 命令以后，我们可以观察到导入 HBase 中的数据（HBase 底层依赖 HDFS）。

为了查看 HBase 中的数据，可使用 HBase shell，为了启动 HBase shell，需要在 VM（CentOS）的 shell 中执行下面的命令：

```
${HBASE_HOME}/bin/hbase shell
```

一旦 HBase shell 初始化完毕，执行图 5-19 中的命令。

在 Sqoop 中，并没有直接的办法从 HBase 中导出数据；然而可以利用一个基于这些 HBase 表的 Hive 视图来实现该目标。在下一节中将深入探讨如何利用 Sqoop 从 HDFS 中导出数据。

非卖品，仅供非商业用途或交流学习使用


```
hbase(main):002:0> scan 'customer'
ROW                                    COLUMN+CELL
 0                                     column=h_cust_col1:dob, timestamp=1487190691627, value=1985-10-20
 0                                     column=h_cust_col1:first_name, timestamp=1487190691627, value=tomcy
 0                                     column=h_cust_col1:last_name, timestamp=1487190691627, value=john
 1                                     column=h_cust_col1:dob, timestamp=1487190691627, value=1989-08-15
 1                                     column=h_cust_col1:first_name, timestamp=1487190691627, value=rahul
 1                                     column=h_cust_col1:last_name, timestamp=1487190691627, value=dev
 2                                     column=h_cust_col1:dob, timestamp=1487190691627, value=1982-08-10
 2                                     column=h_cust_col1:first_name, timestamp=1487190691627, value=pankaj
 2                                     column=h_cust_col1:last_name, timestamp=1487190691627, value=misra
 3                                     column=h_cust_col1:dob, timestamp=1487190691627, value=1990-05-06
 3                                     column=h_cust_col1:first_name, timestamp=1487190691627, value=devi
 3                                     column=h_cust_col1:last_name, timestamp=1487190691627, value=lal
 4                                     column=h_cust_col1:dob, timestamp=1487190691627, value=1992-06-25
 4                                     column=h_cust_col1:first_name, timestamp=1487190691627, value=john
 4                                     column=h_cust_col1:last_name, timestamp=1487190691627, value=doe
5 row(s) in 0.0940 seconds

hbase(main):003:0>
```

图 5-19　在 HBase shell 中查看导入 HBase 中的数据表

5.6.6　Sqoop 数据导出

与 Sqoop 的数据导入功能类似，Sqoop 也可以从 HDFS 中导出数据到关系数据库中去，前提是目标数据库中必须有对应的表：

```
sqoop export (generic-args) (export-args)

sqoop export --connect
jdbc:postgresql://<DB_SERVER_ADDRESS>/<DB_NAME>?schema=<SCHEMA> --table
<TABLE_NAME> --m 1 --username <DB_USER_NAME> --password <DB_PASSWORD> --
export-dir <HDFS_SOURCE_PATH>
```

这里的 <HDFS_SOURCE_PATH> 是 HDFS 文件系统路径，它作为数据源，其中的数据被导出到目标数据库中去，如图 5-20 所示。

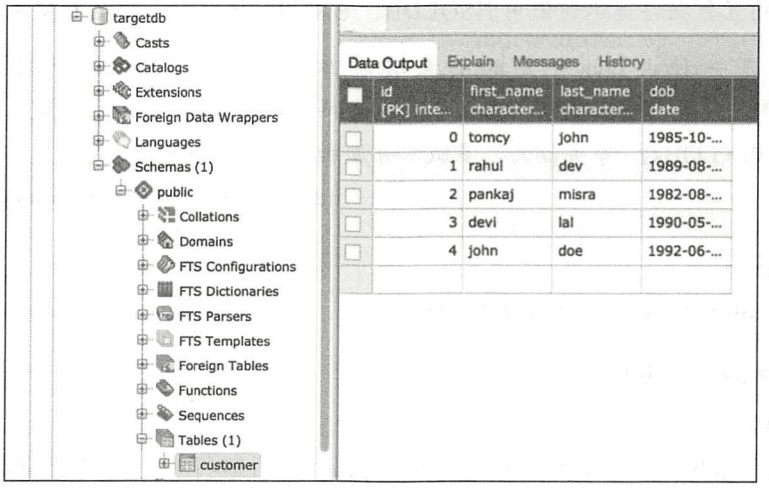


图 5-20　使用 Sqoop 从 HDFS 中导出数据到 RDBMS 中

非卖品，仅供非商业用途或交流学习使用

5.6.7 Sqoop job

Sqoop 也支持 job 的概念。Sqoop job 可以被定义和保存，并且能被 Sqoop 命令重复使用。典型的 Sqoop job 由数据源、目的地定义及少量其他可选参数构成。Sqoop job 被保存在 Sqoop 的元数据中，用于重复使用及远程访问。

1. job 命令

Sqoop 中有一个元数据库，每个 job 作为一个元数据实体（metadata entity）存储在其中。创建 job 的主要目标是一次性定义源系统及目标系统，然后重复利用它执行数据导出与导入。与任意其他实体类似，job 除了支持诸如 create、get、update、delete 之类的生命周期方法（lifecycle method），还支持一些其他功能。我们将在后续的小节中介绍这些功能。

2. 创建 job

典型的 Sqoop job 创建语法如下：

```
sqoop job (generic-args) (job-args)
   [-- [subtool-name] (subtool-args)]

bin/sqoop job --create <JOB_NAME> -- <OPERATION_NAME> --connect
jdbc:postgresql://<DB_SERVER>/sourcedb?schema=public --table <TABLE_NAME> -
-m 1 --username <DB_USER> --password <DB_PASSWORD> --as-avrodatafile

Where,
<JOB_NAME>                  → 将要创建的 job 名
<OPERATION_NAME>            → Sqoop 操作名，如 import
<DB_SERVER>                 → DB Server 地址或 IP
<TABLE_NAME>                → 想要导入的数据表名
<DB_USER>                   → 包含要导入数据的 DB 的用户名
<DB_PASSWORD>               → 包含要导入的数据的 DB 的密码
```

请观察“--”附近的空格。这些空格被解释为命令之间的分隔符。简单起见，嵌入式的内置数据存储不妨使用 HSQLDB，它被用作 Sqoop 1 的元数据库。

HSQLDB 是一种由 Java 开发的关系数据库，提供了 JDBC 驱动器，符合 ANSI SQL:2011 标准。它是一个小型的、高性能的、支持多线程的引擎和服务器，具有内存表和磁盘表、大对象（LOBs）、事务隔离、多版本并发和 ACID 等功能或特性。

—— https://sourceforge.net/projects/hsqldb/

3. 列举 job

配置好的 job 可以使用下面的命令来列举：

```
bin/sqoop job --list
```

4. 运行 job

如果想运行 job，则需要先读取已存储的 job，然后运行它，这里也需要元数据连接（连接到 Sqoop 1 的元数据库，元数据存储在元数据库中）。下面的命令用来执行一个保存好

非卖品，仅供非商业用途或交流学习使用

的 job：

```
bin/sqoop job --exec <JOB_NAME>
```

如果提示要输入密码，那么请输入 PostgreSQL 用户密码。

5.6.8　Sqoop 2

Sqoop 2 拥有前面描述过的 Sqoop 1 的所有能力，同时也增添了很多 Sqoop 1 不具备的新能力。这些新能力包括 Sqoop shell，用于支持多种数据库的元数据仓库，链接（link）概念，及 Sqoop job 的远程执行能力。

Sqoop 2 在一个嵌入式容器中运行，依赖 Hadoop 环境变量来定位所有的依赖。

这里，我们说 Sqoop 2 指的是 Sqoop 2 的 v5 版本，主要考虑到它工作效果比撰写本书最新时的 Sqoop 2 稳定版本略好。

为安装 Sqoop 2，需要执行下列安装步骤：

1）使用下面的命令下载 Sqoop 1.99.5（官方称之为 Sqoop 2）的二进制安装包，该版本适用于 Hadoop 2：

```
wget
http://archive.apache.org/dist/sqoop/1.99.5/sqoop-1.99.5-bin-hadoop
200.tar.gz
```

2）一旦下载完成，使用下面的命令将安装包解压到用户目录中去。假设解压目录为 ${SQOOP2_HOME}，相应地，对 ~/.bashrc 文件进行更新。

```
tar -zxvf ${DOWNLOAD_DIR}/sqoop-1.99.5-bin-hadoop200.tar.gz
```

3）对 ${SQOOP2_HOME}/server/conf/catalina.properties 文件进行配置，使之包含 Hadoop 安装相关的所有 classpath 的绝对路径。下面是相关的配置范例，我们使用 ${HADOOP_HOME} 来引用 /data1/home/centos/hadoop-2.7.3，这里需要使用绝对路径，以防止系统不能正确地解析环境变量：

```
common.loader=${catalina.base}/lib,${catalina.base}/lib/*.jar,${cat
alina.home}/lib,${catalina.home}/lib/*.jar,${catalina.home}/../lib/
*.jar,/data1/home/centos/hadoop-2.7.3/share/hadoop/common/lib/*.jar
,/data1/home/centos/hadoop-2.7.3/share/hadoop/common/*.jar,/data1/h
ome/centos/hadoop-2.7.3/share/hadoop/hdfs/*.jar,/data1/home/centos/
hadoop-2.7.3/share/hadoop/hdfs/lib/*.jar,/data1/home/centos/hadoop-
2.7.3/share/hadoop/mapreduce/*.jar,/data1/home/centos/hadoop-2.7.3/
share/hadoop/mapreduce/lib/*.jar,/data1/home/centos/hadoop-2.7.3/sh
are/hadoop/yarn/*.jar,/data1/home/centos/hadoop-2.7.3/share/hadoop/
yarn/lib/*.jar
```

4）配置 ${SQOOP2_HOME}/server/conf/sqoop.properties，使之指向 Hadoop 配置，如下所示：

```
org.apache.sqoop.submission.engine.mapreduce.configuration.director
y=/data1/home/centos/hadoop-2.7.3/etc/hadoop
```

仅供非商业用途或交流学习使用

5）使用下面的命令设置 Sqoop 2 的执行权限：

```
chmod +x ${SQOOP2_HOME}/bin/*
```

6）运行下面的命令来确定 Sqoop 2 是否配置成功（如果遇到 java.sql.SQLNonTransientConnectionException: No current connection 之类的错误，请忽略，此时的配置已经完全适用了）：

```
${SQOOP2_HOME}/bin/sqoop2-tool verify
```

7）运行下面的命令启动 Sqoop 2 服务：

```
${SQOOP2_HOME}/bin/sqoop2-server start
```

设计 Hue 时就考虑了与 Sqoop 2 集成，不过在本书撰写之时，Hue 与 Sqoop 2 之间还存在很多不兼容问题。这些不兼容问题在 JIRA 中被标记为 bug。可以查看 https://issues.cloudera.org/browse/HUE-5128 了解此类问题。

我们可以观察到，Sqoop 2 已设置完毕，并且能正常工作。此时可以使用 Sqoop 2 的 shell 了，同时也可以在 Hue 中查看配置好的 job 与 connector。下面提供了一些截图，读者可以作为启发式的参考材料。如果确定使用 Hue（适用于 Sqoop 2），我们将看到 Hue 拥有对 Sqoop 2 的强大的动态管理能力。

图 5-21 中，有一些资源连接的配置，可以通过 Hue 来对 Sqoop 2 进行配置（在 Data Browser 菜单中选择 Sqoop Transer 菜单项）。在这里，我们可以配置数据库连接及 HDFS 连接。

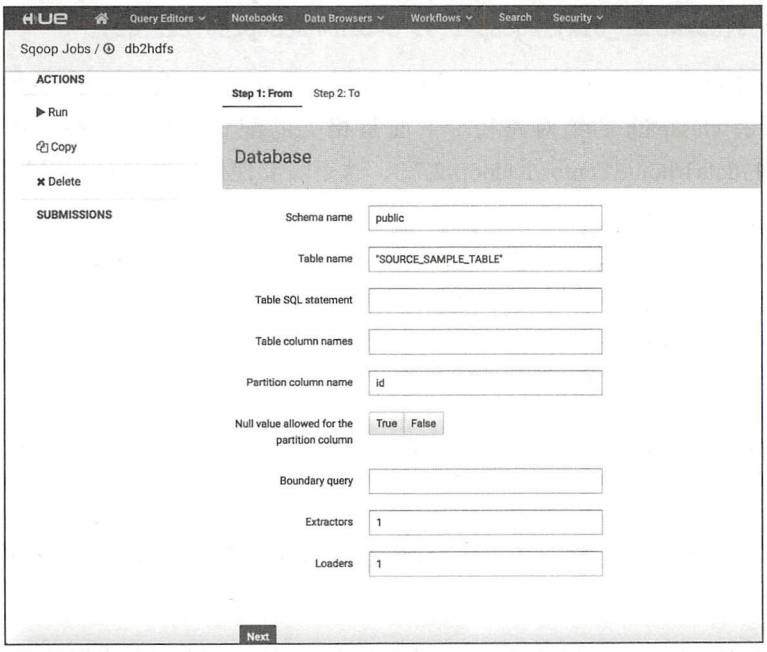


图 5-21 Hue、Sqoop 2 中的数据库、HDFS 的配置

仅供非商业用途或交流学习使用

单击 Hue 中的 Next 按钮，跳转到下一页如图 5-22 所示，在这里设置数据库连接配置的细节，最后保存配置，这样该数据库连接的配置将被保存到元数据库（database repository）中。

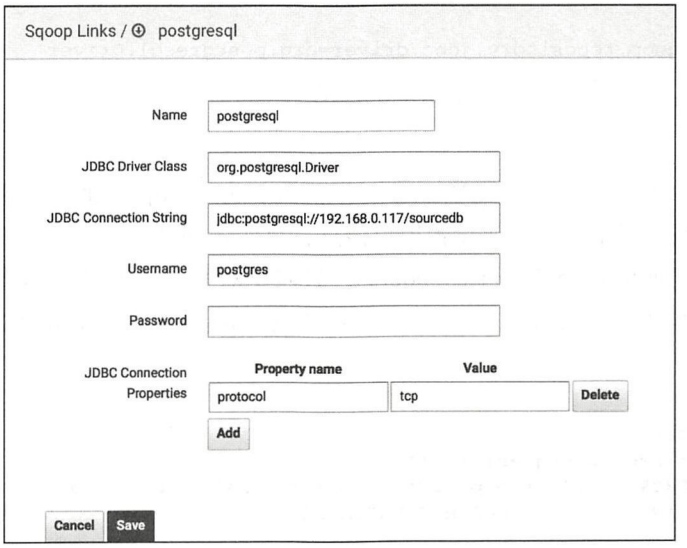


图 5-22　RDBMS 连接

类似地，也可以配置 HDFS 连接，配置时需要 hdfs 连接名以及 HDFS URI，如图 5-23 所示。

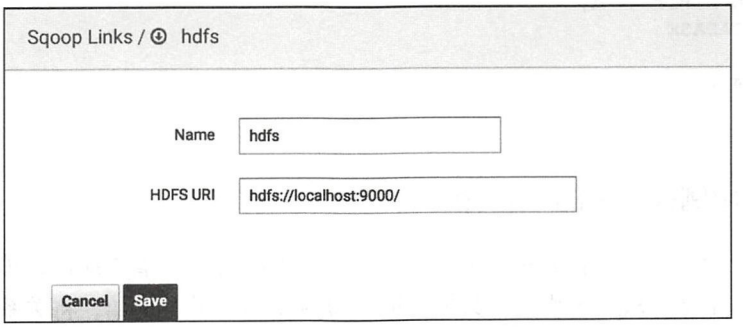


图 5-23　HDFS 连接

与 Sqoop 1 类似，Sqoop 2 也需要一个元数据库。不过默认情况下，Sqoop 2 使用一个叫作 Apache Derby 的嵌入式数据库（https://db.apache.org/derby/）；外部数据库也可以用作元数据库，这可以通过 sqoop.properties 来配置。下面的代码块是该文件配置的一个例子，给出了 Sqoop 2 repository 的相关配置：

```
# 外部 connector 加载路径
# "/path/to/external/connectors/": 将所有的 connector JAR 包添加到特定文件夹
```

仅供非商业用途或交流学习使用

```
#
org.apache.sqoop.connector.external.loadpath=
org.apache.sqoop.repository.jdbc.handler=org.apache.sqoop.repository.postgr
esql.PostgresqlRepositoryHandler
org.apache.sqoop.repository.jdbc.maximum.connections=4
org.apache.sqoop.repository.jdbc.url=jdbc:postgresql://192.168.43.28/sqoop?
schema=SQOOP
org.apache.sqoop.repository.jdbc.driver=org.postgresql.Driver
org.apache.sqoop.repository.jdbc.user=sqoop
org.apache.sqoop.repository.jdbc.password=sqoop
org.apache.sqoop.repository.jdbc.transaction.isolation=READ_COMMITTED
```

如上面的代码所示，为了让配置正常工作，需要为 Sqoop 创建一个用户。Cloudera 提供了一个官方文档，用于指导如何设置用户信息，可参考网址 https://goo.gl/F6iJsb。

也可以参考下面的指令（来自 Cloudera），只需要对 PostgreSQL 版本部分做少量修改，就可以正常运行本书中的范例了。完整的安装步骤及 Sqoop 的细节超出了本书的介绍范围，此处不过多介绍。

```
$ psql -U postgres
Password for user postgres: *****
postgres=# CREATE ROLE sqoop LOGIN ENCRYPTED PASSWORD 'sqoop'
NOSUPERUSER INHERIT CREATEDB NOCREATEROLE;
CREATE ROLE

postgres=# CREATE DATABASE "sqoop" WITH OWNER = sqoop
ENCODING = 'UTF8'
TABLESPACE = pg_default
LC_COLLATE = 'en_US.UTF-8'
LC_CTYPE = 'en_US.UTF-8'
CONNECTION LIMIT = -1;
CREATE DATABASE

postgres=# q
```

5.6.9 SCV 用例视角中的 Sqoop

读者已经看到了，Sqoop 实际上是构建单一客户视图的一个重要部件，单一客户视图概念可参考本书第 1 章。Sqoop 覆盖了数据获取的最主要的渠道，例如，将关系数据库中的数据传输到 HDFS 中。大多数业务和合作伙伴应用程序都属于这一类别，大部分是关于结构化信息的。

因此，从构建单一客户视图的角度来看，Sqoop 如图 5-24 所示。

从图 5-24 可以看到，在整个单一客户视图之外，依然还有多种类型的信息可以被捕获，相关内容将在后面数章中进行介绍。当我们开始为单一客户视图构建一个完整的数据湖时，可以可视化方式来介绍各种架构层级和组件的引入。

从对 Sqoop 使用的角度来看，这里引入功能层级为数据获取层，如图 5-25 所示。

仅供非商业用途或交流学习使用

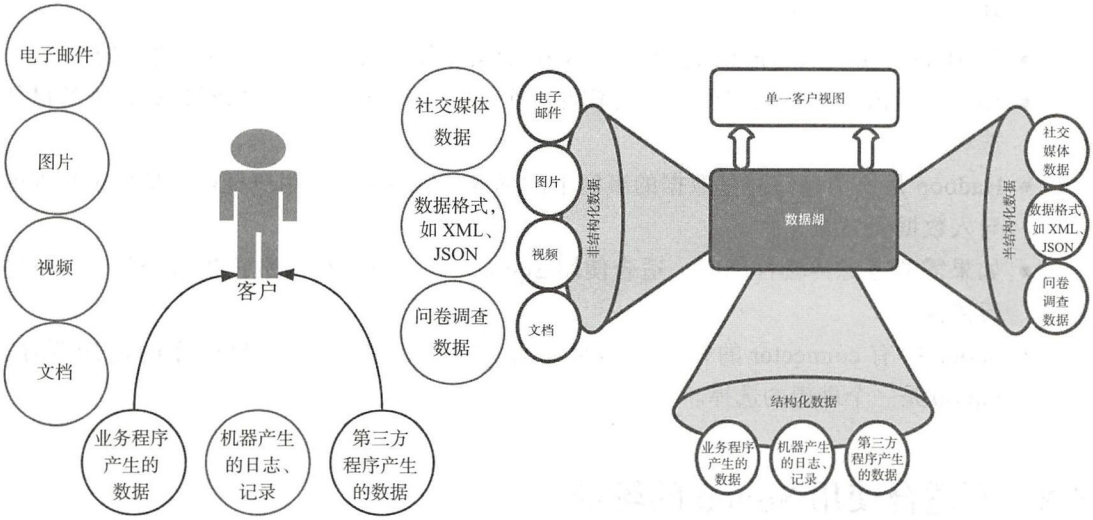


图 5-24　单一客户视图案例中的 Sqoop

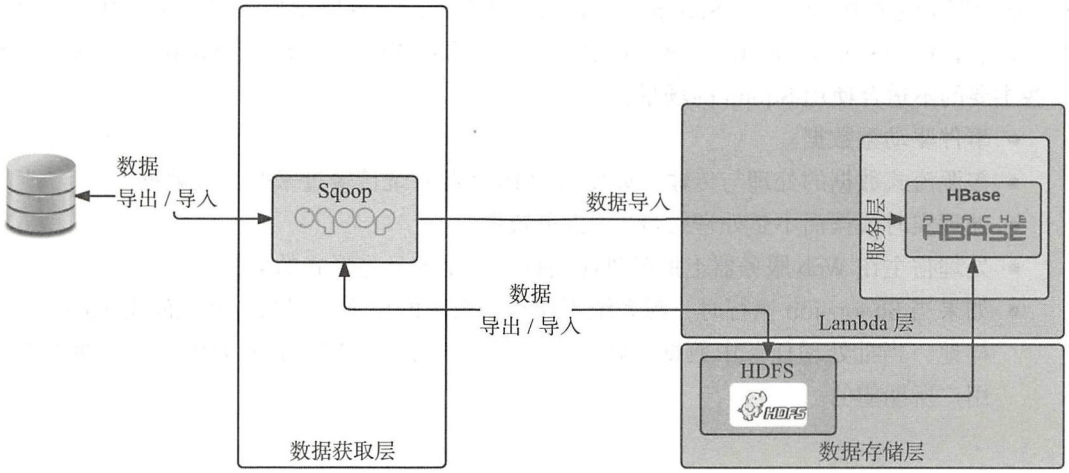


图 5-25　单一客户视图中的数据获取层与 Sqoop

5.7　适合使用 Sqoop 的场景

Apache Sqoop 可以用来满足数据湖中的多种数据传输需求，这些场景中，HDFS 作为主要的数据存储手段，用于存储来自各种源系统的数据。下面列举了一些案例，在这些场景中使用 Sqoop 更有意义：

- 对于 RDBMS 与 Hadoop（HDFS/Hive/HBase）数据传输，例如常规的批处理，或近似批处理，比较适合使用 Apache Sqoop。Apache Sqoop 是数据获取层的一种主要的

仅供非商业用途或交流学习使用

被广为使用的工具。

- 从 MongoDB、Cassandra 之类的 NoSQL 数据库传输数据到 Hadoop 文件系统中。
- 拥有大量依赖关系数据库的应用软件的企业，Sqoop 是传输数据到数据湖中的最佳选项。
- Hadoop 是一个存储海量数据的事实上的标准。Sqoop 允许用户轻松地从传统数据库导入数据到 HDFS 中。
- 如果需要考虑传输性能时，适合使用 Sqoop，因为它可以对数据进行分割，并行进行传输。
- Sqoop 中有 connector 的概念，如果企业中有多种依赖不同数据库的应用程序，Sqoop 是一个理想的选择。

5.8 不适合使用 Sqoop 的场景

当业务极度依赖如 Oracle、MySQL、PostgreSQL、Teradata 之类的关系数据库时，Sqoop 是最合适的配套工具。Sqoop 并不是特别适合处理事件驱动型数据。对于事件驱动型数据，适合使用 Apache Flume（第 6 章中将深入介绍此类应用）而不是 Sqoop。下面列举了一些主要的不适合使用 Sqoop 的场景：

- 事件驱动型数据。
- 多源流式数据的处理与传输，如使用 JMS 从源系统传输过来的流式数据。
- 处理实时数据而不是处理批量 / 小批量数据。
- 处理宿主在 Web 服务器上的各种应用程序产生的日志形式的数据。
- 如果当 Sqoop job 执行时，源系统不能承受较大的压力，那么应避免使用 Sqoop。同样地，当批处理任务中数据量特别大时，会给源系统带来更大的压力，这通常不是用户所期望的。

5.9 实时 Sqooping 是否可行

对于实时数据摄取，Sqoop 并不是一个很好的选择。对于不短于 5 分钟的近实时处理（这里的 5 分钟只是用来举例，并没有特殊含义），则可以使用 Sqoop 来传输数据。因为此类用法操作频率更快，因此待处理的数据量需予以控制，使得 Sqoop 能在下一次任务执行启动之前将本批次数据处理完。

5.10 其他选项

从关系数据库向 Hadoop 文件系统批量传输数据，开源社区中并没有太多选择。然而，

仅供非商业用途或交流学习使用

还是有一些其他的可用的工具来完成此类任务，在本节中将会向读者介绍这些选项。读者可根据企业的实际需求来评估这些技术是否适合作为数据湖的组件。

5.10.1　原生大数据 connector

大多数主流关系数据库都提供了 connector，供关系数据库与 Hadoop 文件系统之间导入或导出数据。如果使用的是 Oracle 数据库，它提供了一整套工具用于 Oracle 数据库与 Apache Hadoop 之间的数据集成。图 5-26 显示了完整的 Oracle 大数据 connector 产品套件，以及它们能提供的功能（细节请参考 http://www.oracle.com）。

与 Oracle 类似，MySQL 关系数据库则提供了 MySQL Applier，它是一个用于从 MySQL 向 Hadoop 文件系统导入数据的原生大数据 connector。MySQL Applier 也可以提供增量数据传输（实时）能力，这与 Sqoop 提供的传统的批处理能力不同。图 5-27 展示了 MySQL 用于向 HDFS 传输数据的原生 connector。

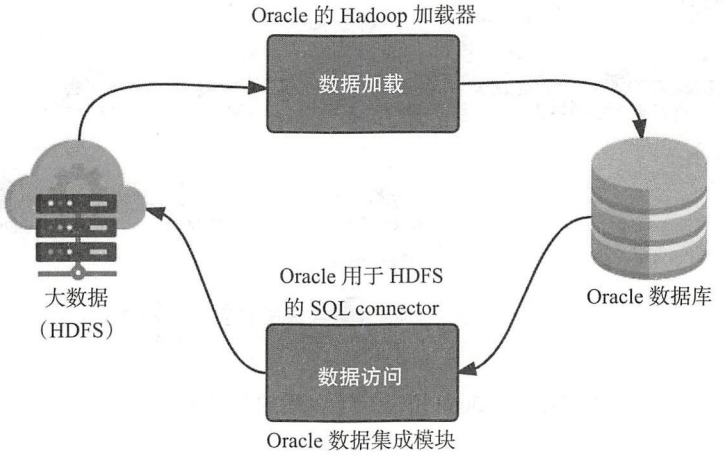


图 5-26　Oracle 大数据 connector 产品套件

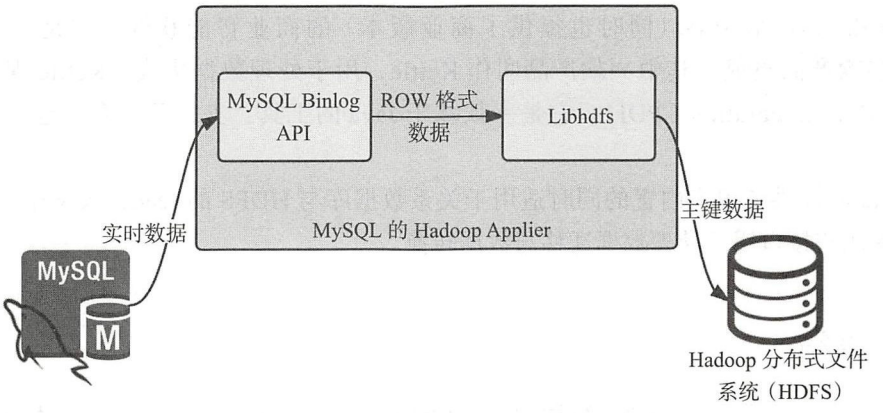


图 5-27　适用于 Hadoop 的 MySQL Applier

仅供非商业用途或交流学习使用

5.10.2 Talend

Talend 是一款开源的 ETL 开发（图形化的）、监控、调度工具。就本章的目标而言，我们仅讨论 Talend 的 ETL 相关功能，实际上，Talend 是一个产品套件，其中很多功能适用于大数据处理场景。Talend 有强大的开源社区的支持，它提供了海量的 connector 支持（超过 800 种，是已知最大的 connector 库），利用 Talend 提供的各种工具，用户能轻松完成数据集成任务。Talend 是一款成熟的产品，支持多种大数据技术。

由于提供了丰富的 connector，Talend 可以在多种关系数据与 Hadoop 之间轻松地集成和传输数据，它是 Sqoop 之外的另一个选择。Talend 同时也提供了图形用户接口，在这里用户可以编写和执行数据流水线（data pipeline），这种操作方式对用户非常友好。Talend 也提供了 Sqoop connector，因此也可以利用 Sqoop 在大数据处理方面的一些优势。

图 5-28 是来自 talendexpert.com 的数据转换参考图。

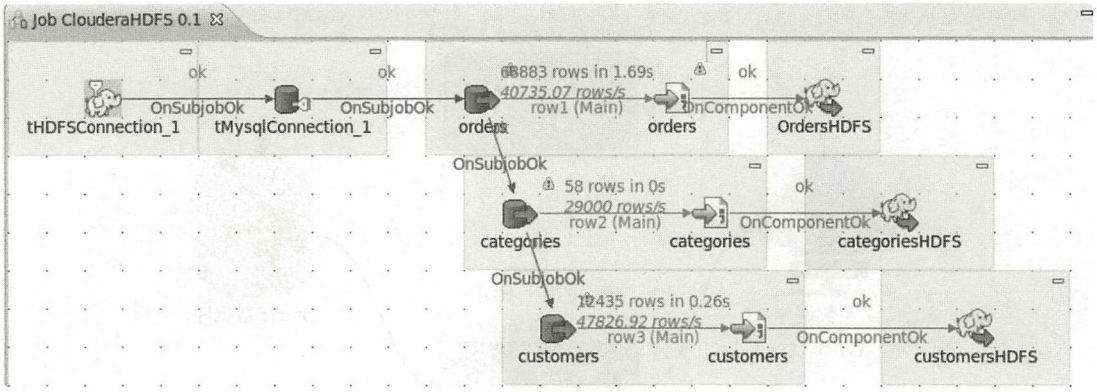


图 5-28 Talend 用于 ETL 开发的图形用户接口

5.10.3 Pentaho Kettle（PDI——Pentaho 数据集成）

Pentaho 是一款开源（同时也提供了商业版本）的商业智能软件，它是一个产品套件，由多款产品组成，其中一款产品叫作 Kettle，用于处理数据集成。Kettle 现在被称为 Pentaho Data Integration（PDI）。它是一款基于 Java 的工具，支持跨平台，也支持多种数据库。

Pentaho 提供了很多内置的同时适用于关系数据库与 HDFS 的 reader 及 writer，同时提供了图形用户接口用于处理数据迁移与数据转换。

5.11 总结

从本章起，我们开始介绍与数据湖每个功能模块 / 层相对应的具体技术。本章中介绍的

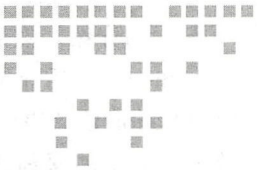
仅供非商业用途或交流学习使用

是与数据获取层相关的技术。最开始介绍了数据获取层的定义，然后介绍了使用 Sqoop 的理由，以及该工具的优缺点。之后对 Sqoop 及其架构做了深入的介绍。然后介绍了 Sqoop 的两个重要版本：Sqoop 1 及 Sqoop 2。首先介绍理论层面的知识，然后介绍 Sqoop 实际应用，如系统安装、系统运行等，最后介绍在 SCV 用例中使用 Sqoop 能获得什么。

通过阅读本章内容，读者可以获得对数据湖架构中数据获取层的深刻认识，同时读者也对 Apache Sqoop 有了足够深入的理解，明白了为什么选取 Sqoop 作为数据获取层的处理组件。当然，读者也通过本章掌握了 Sqoop 的内部运行机制及实操细节。如果读者亲自实践过书中范例，那么就已经知道如何利用 Sqoop 构建数据湖的数据获取层，以及在 PostgreSQL 与 HDFS 间传输数据了。本章真正开启了我们的数据湖 SCV 用例之旅。

下一章将介绍数据获取层中的另一个技术框架——Apache Flume，届时将深入介绍该框架的底层原理与实操细节，介绍方式将与本章类似。

仅供非商业用途或交流学习使用



Chapter 6 第 6 章

基于 Apache Flume 的流式数据获取

为了继续探究数据湖中的各种技术和功能模块 / 层，本章开始介绍数据获取层中使用的另一项技术。与前一章（实际上是本书这一部分的其他所有章节）类似，首先从数据湖的总体架构入手，然后深入研究所选择的技术。

在深入研究这项技术之前，我们将给出选择该技术的理由，并且向读者详细介绍这项技术，在读者对技术细节有了充分的了解后，就可以在企业中实际应用这些技术。

本章会探讨数据获取层中使用的第二项技术——Apache Flume。我们将由浅层次入手介绍 Apache Flume，然后深入探讨该技术本质，最后将结合 SCV 用例，向读者展示一个实际的 Flume 应用案例。本章的最后一部分是为了让读者熟悉其他类似的技术，这些技术可以用来代替 Flume 来实现数据湖架构中对应的功能。

阅读完本章后，读者将会对 Flume 在架构中的使用情况有一个清晰的认识，同时也会对 Flume 的技术细节有充分的了解。读者也能获得 Flume 的实践经验，这些经验在构建数据湖及实现 SCV 用例的过程中还会进一步强化。

这真的很刺激，让我们开始吧！

6.1 数据获取

大数据的 4V 之一就是处理速度（velocity），正是由于这个 V，本章介绍的数据获取技术在现代企业运营中更显得意义重大。传统模式中，分析都是在数据收集完毕（缓慢变换的数据）的基础上完成的，但是如今，分析通常是基于实时的流式数据进行的，借此及时采取措施来调整业务。比如在用户的 Twitter 流上进行回应来提升用户体验，或者通过观察用户近期

仅供非商业用途或交流学习使用

在网站上的行为来提供个性化的推荐。本章将主要介绍数据湖中的实时数据获取部分。

在第 5 章中已经详细介绍了数据获取层，本节就不再赘述了。然而，第 5 章介绍的批量数据和本章要介绍的实时数据存在着显著的差异。在深入研究技术方面之前，读者需要先详细了解这种类型的数据。

6.1.1　什么是流式数据

流式数据是指由各种业务应用和外部应用（几乎当前所有的社交媒体）连续而快速地生成的数据，通常情况下负载较小。很多类型的数据都属于这个范畴，例如：

- 各种 Web 和应用服务器生成的日志数据。
- 企业网站的用户行为数据（网页展示、链接点击等）。
- 客户在各种社交网络平台上产生的大量数据。
- 来自各种传感器的数据（由于近年来物联网兴起）。

由于实时数据是逐条抵达的，因此按顺序处理是有意义的。对于企业来说，分析这些数据然后做出适当的回应，这是一种业务模式，同时也改变了企业的工作方式。实时分析这些数据，然后根据客户需求提供个性化服务，这是一种双赢的局面，既可以为企业带来收益，也可以改善客户体验（无形收益）。

图 6-1 显示了来自各种数据源的流式数据，这些数据最终可以直接进入数据湖，或者经过处理后被存储起来。

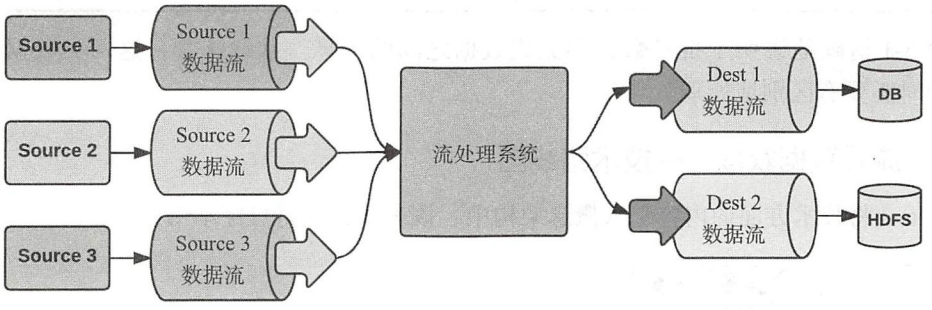


图 6-1　数据流实战场景

企业应该开始使用这些数据，实现一些最低限度的功能，比如说，可以收集日志数据，来查看是否存在可以主动发现和纠正的错误。经过一段时间的分析，可以实现复杂的功能，比如分析用户行为数据（用户在网站的行为数据）就可以实时为用户提供适当的推荐。

最好将这些数据流式写入 Hadoop 文件系统中，而不需要任何处理，当然也可以对数据进行一些在业务上有价值的处理。处理后的数据可以流入可视化分析工具，然后展示各种必要的业务细节，以便实时对业务决策提供支持。比如说，分析电商网站的订单数据，如果没有达到预期目标，则可以对商品价格进行必要的调整（比如对库存量大的商品进行打折）。

分析流式数据时，一个极其重要的问题是如何按照收到消息的顺序来处理消息。如果

顺序不能保证，分析可能导致错误的结论，这会对业务产生副作用。获取的数据量会不断变化，当数据涌入的速度非常快时，系统各个组件的负载也会激增。在设计架构时应充分考虑系统的可扩展性、持久性及其他方面。

6.1.2 批量数据和流式数据

表 6-1 中总结了批量数据和流式数据之间存在的主要差异。我们的用例确实包含了这两种类型的数据。为了处理批量数据，我们选择并使用 Sqoop，而为了处理流式数据并写入 Hadoop，则采用了 Flume。

表 6-1 批量数据与流式数据差别对比

批量数据	流式数据
数据量相当大，在规定的时间内处理大量的数据。批量数据由成千上万条记录组成	数据量不大，但是需要快速处理，数据持续到达，并且没有确切的到达周期。流式数据由单条或少量记录组成。少量记录有时也称为微批量（micro-batch），因为与传统的批量数据相比，这种批量中的记录非常少
分析或处理是在大周期数据或者全量数据上进行的	在移动时间窗口中以连续的方式对小数据集进行分析和处理
操作周期从几分钟到几个小时不等	在大多数情况下，操作周期从毫秒到秒不等
由于数据集庞大，可以进行复杂的分析，通常可以支持商业决策	可以进行简单分析，在通常的情况下和批量分析一起来获取信息
示例：上个月促销时，用户购买了更多的电子产品	示例：当电商网站产生订单时进行信用卡欺诈检测或者对重要交易进行风险分析（Risk Based Analysis，RBA）。根据风险，加强多因素认证（如一次性密码）的机制

表 6-1 清晰地说明了批量数据和流式数据之间的差异。这意味着，这两类数据在进入数据湖时需要有区别地处理。

6.1.3 流式数据获取——技术路线图

图 6-2 将技术方面的内容引入概念架构中，该表述方式将贯穿本书。

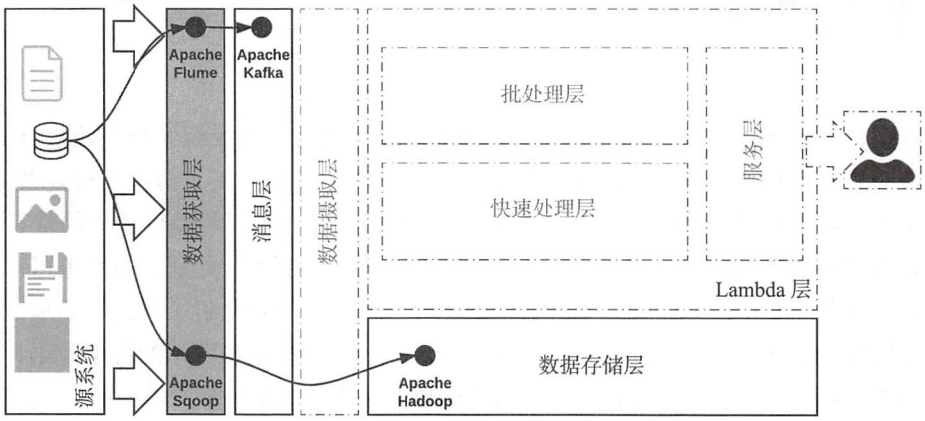


图 6-2 数据获取层技术路线图

我们选择了用 Apache Flume 作为实时数据传输技术，它位于数据湖的数据获取层。

与 SCV 用例一样，来自各种业务应用程序的实时数据将流入 Flume，然后传输到 Hadoop 文件系统，并进行后续分析。这些实时数据是用户在企业网站中的行为数据。诸如页面访问、链接点击、位置细节、浏览器细节等，这些数据将流入 Flume，然后存储在 HDFS 中。

图 6-3 显示了本章将要深入探讨的内容，而数据湖的其他方面的内容都故意从图中隐去了。但是图中还是包含了 Sqoop，我们正在按照本书内容编排顺序逐步构建完整的数据湖架构。

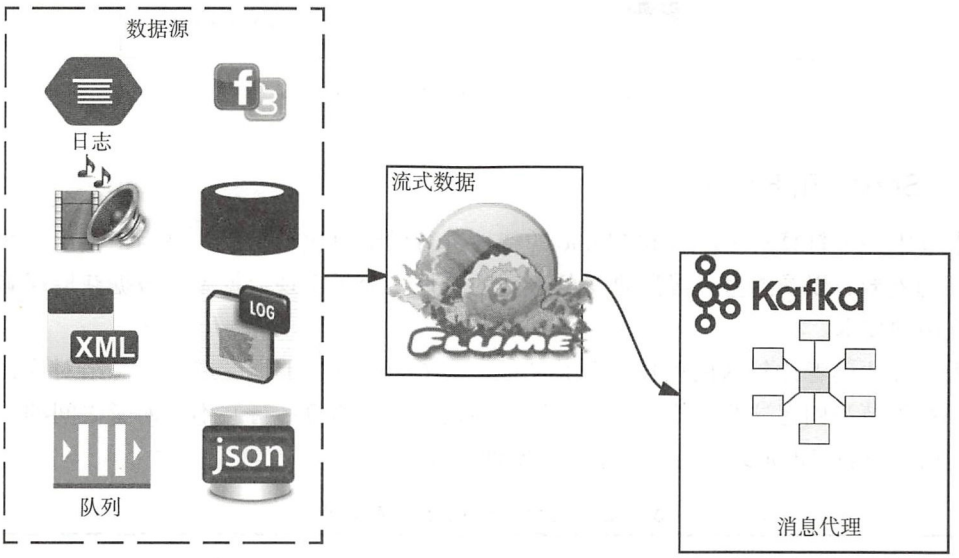


图 6-3　数据湖中的 Flume

6.1.4　什么是 Flume

Flume 的本意（非技术概念）是通过修建渠道，利用渠道的引流能力来从山上运输木材。这些渠道可以实现远距离、快速、高效且低成本地运输木材。虽然最初的主要目的是运输木材，但是这些渠道其实可用于传输各种各样的材料。

正如 Flume 这个名词的本意，这项技术可以从数据源传输大量数据到目的地。与物理上的水槽类似，Flume 这项技术也是为了传输累积在各个服务器（或者其他软件组件）中的日志数据而构建的，并将这些数据聚合起来以供整合分析。后来，它被扩展以支持不同的数据源和目的地。

图 6-4 从概念上显示了 Flume 的功能。在本书的案例（基于数据湖的 SCV）中，我们将获取日志数据（从服务器收集的）和事件数据（主要是行为数据），将其导入公共的存储系统中（Hadoop 系统），如之前图 6-3 所示。

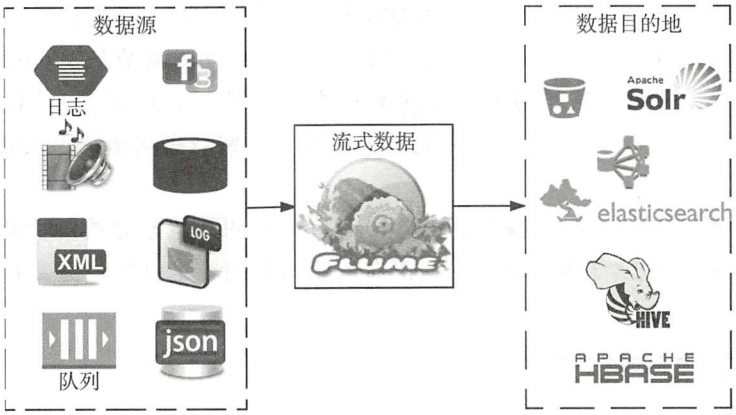


图 6-4 Flume 功能概念视图

6.1.5 Sqoop 和 Flume

本节中并不打算对 Sqoop 和 Flume 的特性进行详细对比，相反，是为了突出这两种技术之间的差异，以及这些差异所带来的不同的特性。为了实现数据湖，数据获取层确实都需要这两种技术。

为了方便读者比较这两项技术及其功能，表 6-2 中对若干维度进行了总结对比，并提供了一些细节来阐述这几个维度。需要注意的是，表格的每一行并不是两者之间的一对一比较，而是以表格的形式进行了总结，方便读者阅读。

表 6-2 Sqoop 与 Flume 差别概览

Sqoop	Flume
批量数据处理	流式数据（实时数据）处理
处理大量数据，数据量可以达到 GB 级到 TB 级	处理高速数据，数据以消息的形式传输，数据量在 KB 级到 MB 级之间
主要功能是把数据从 RDBMS 导入 Hadoop（HDFS）	主要功能是把流动的数据（流）传输到 Hadoop（HDFS）
兼容各种 RDBMS，主要使用 JDBC 技术。RDBMS 的供应商提供了各自的 connector 工具，并可以使用各种优化技术来实现最佳效果	除了消息类型的数据，Flume 也能很好地将日志数据引入 Hadoop
能够并行地进行数据传输	N/A
特别适合处理静态数据（数据驻留在 RDBMS 中）	特别适合处理流动数据
作用于本质上（或多或少）是静态的数据	作用于经常变化的数据。由于是事件驱动的，所以这些事件到达的顺序非常重要，必须确保在进入 Hadoop 环境时能保持数据完整性
作用于已经聚合并归集到一起的数据	可以收集和汇总数据，处理近期到达的数据
	天然具有可靠性、高可用性和分布式
批量处理，可能给向 HDFS 传输数据的数据源系统造成不必要的压力	不会对数据源系统增加负载，以非连接方式工作

非卖品，仅供非商业用途或交流学习使用

图 6-5 很好地总结了在实现数据湖的需求中，Sqoop 和 Flume 所能提供的功能。

6.2　为什么使用 Flume

本节将专门解释选择 Flume 作为技术解决方案的原因，我们希望基于它来构建处理流式 / 实时数据的数据获取层。

下一个小节将首先介绍 Flume 的历史，然后介绍它的优缺点。其优点是我们选择这种技术来处理将实时数据传输到 Hadoop 中的主要原因。

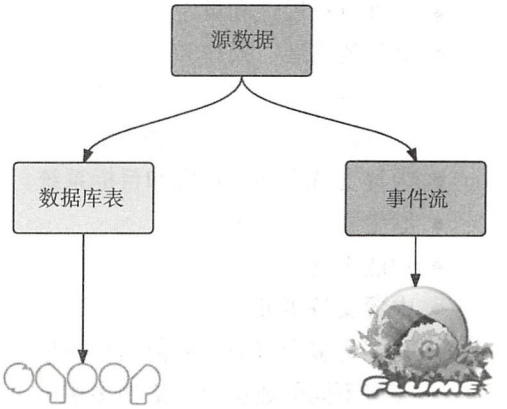


图 6-5　Sqoop 与 Flume

6.2.1　Flume 简史

Apache Flume 由 Cloudera 公司开发，专门用于处理和迁移海量数据到 Hadoop。企业都希望能够以无延迟或者极低的延迟（NRT：实时或近实时）的方式将生成的数据转移到 Hadoop 系统，以便进行各种分析。借助于 Flume，这种美妙的架构就得以实现了。

如前一节所述，最初设计和开发 Flume 是为了专门从各种数据源（Web 服务器）收集和聚合日志数据并写入 Hadoop，以便执行各种分析并进而对系统进行主动维护。后来 Flume 又经过了重新设计和重构，能够兼容各种不同的数据源和目的地，同时在设计中考虑了可插拔性（pluggability）和可扩展性（extensibility）。

与 Sqoop 类似，Flume 也有两种主要的版本，如下所示：

- Flume OG（Old Generation：1.0 以前的版本）
- Flume NG（New Generation）

顾名思义，Flume OG 是 Flume 最初的发行版，然后经过了整体的重写和重构，产生了 Flume NG。Flume NG 是当前活跃的和持续维护的 Apache 项目，也是本书将使用的版本。在撰写本书时，Flume NG 最新的版本是 1.7.0，发布于 2016 年 10 月 17 日。而 Flume 加入了 Apache 孵化器时的版本为 1.0.0 和 1.1.0。

6.2.2　Flume 的优势

Apache Flume 的核心优点如下：

- 开源。
- 有很好的文档编制，包含很多示例程序和应用模式。
- 高吞吐量，低延迟。
- 声明式配置（declarative configuration）。

非卖品，仅供非商业用途或交流学习使用

- 天然地支持分布式。
- 高可靠、高可用、可水平伸缩。
- 高度可扩展性和可定制性。
- 极低的安装、操作和维护成本。
- 本节专门有一个小节来介绍上下文相关路由（contextual routing）。但是对读者来说，该方面是关于 Flume 查看数据负载（流式数据或者事件）和构建合适的路由的。
- 内置支持多种数据源和目标系统。
- 原生高度可插拔。
- 功能丰富。
- 内置支持事务。
- 能够将多个服务器的数据轻松地导入 Hadoop。
- 支持不同的数据流，诸如多跳（multiple-hop）、扇出（fan-out）、扇入（fan-in）等。
- 良好的集成或支持各种现有的监控工具。

6.2.3 Flume 的劣势

虽然 Flume 有不少优点，但是在某些方面仍然存在着不足：

- 较弱的顺序保证。
- 不能保证到达消息的唯一性（在很多情况下，可能会出现重复的消息）。
- 对企业而言，为典型的 Flume 应用进行硬件选型是件棘手的事，大多数情况下，需要反复试验。因此，对于它的可扩展性往往需要慎重考虑。
- Flume 的吞吐量取决于 Channel 的后备存储系统。因此，如果在选择后备存储时考虑得不够充分，可扩展性和可靠性就成了问题。

显然，Flume 的优势完全能掩盖了它的不足，这也是选择它的主要原因。另外，Flume 普及率较高（相对的），虽然没有在前面的小节中提及，但这对于企业来说很重要（容易招聘对口的工程师，有良好的社区支持，技术问题可得到快速响应）。

6.3 Flume 的架构原则

任何技术想要获得成功，都应该明确地定义架构原则，然后在这个基础上进行设计并逐步演进。下面是设计 Flume 时遵循的一些架构原则（其中一些是 Flume NG 引入的）：

- 可靠性（reliability）：即便是发生了各种故障（大多数时候是系统部分失效），也能持续接收流式数据和事件而不发生任何丢失。Flume 非常重视的一个核心架构原则是容错性（fault-tolerance），这意味着即使某些组件出错，某些硬件出现故障或者网络带宽性能不佳，Flume 在大多数情况下也能承受这些问题，保证主要功能运转而不是完全关闭。Flume 能保证，只要 Flume Agent 在运行中，到达 Flume Agent 的数据

非卖品，仅供非商业用途或交流学习使用

最终都能被转交给其他组件。一些设置可以用来控制可靠性的级别。显然，可靠性越高，可伸缩性就越低。

- 可伸缩性（scalability）：只需要改变硬件拓扑，Flume 就可以处理更多的流式数据。Flume 的水平可伸缩性体现在，它允许通过添加机器来满足消息吞吐量的增长。在架构部分，本书将涵盖进行水平伸缩时需要改动的各种组件。然而，可伸缩性取决于目标系统从数据流水线中提取数据的能力，有时候这可以用来定义 Flume 可伸缩的程度。
- 可管理性（manageability）：作为解决方案的一部分，能够集中管理架构中的各种组件是能在实际生产中成功应用架构的关键。在 Apache Flume 架构中，运用 Flume Master（下一节将详细介绍）组件，可以通过 Web 界面或者 Flume 命令行来集中地管理所有组件。
- 可扩展性（extensibility）：Flume 遵循的一个非常重要的原则是可以整合各种数据源和目标系统。这是一个强制性的要求，同时明确为 Flume 设计和架构的核心原则之一。实现这条原则主要是通过开发或使用内置的 connector 来连接 Flume 的输入端和输出端。

这些就是构建 Flume 所遵循的核心架构原则，在下一节中，这些内容的很多方面将得到详细介绍。

6.4　Flume 架构

在前面的章节中讨论了构建 Flume 需要遵循的架构原则，现在我们开始深入研究它的架构。首先在图 6-6 中介绍 Flume 的基本架构，然后在下面章节中继续深入研究它。

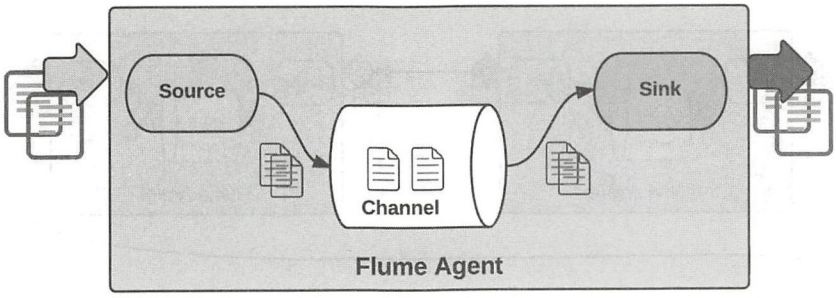


图 6-6　Flume 基本架构

一个简单的 Flume 架构有 3 个重要的组件，它们协同工作，并且以实时的方式将源数据（流式数据或日志数据）传输到目的地：

- Source：用于监听流式数据或者事件，然后将其写入 Channel。
- Channel：一个存储数据的流水线，数据存储在其中直到被其他组件取走。

非卖品，仅供非商业用途或交流学习使用

- Sink：用于从 Channel 中获取事件并进一步处理（发送到另一个 Source）或者持久化到数据存储。如果 Sink 操作失败，它可以自动持续重试，直到成功为止。

表 6-3 所示为 Flume 的各个组件，即 Source、Channel 和 Sink，总结了一些例子。

表 6-3 一些 Source、Channel、Sink 的例子

Source	Channel	Sink
Console	Memory	Memory
Exec	File	JDBC
Syslog	JDBC	File
Avro	…	Null
JMS		HDFS
Spooling Directory		…
…		

图 6-6 中介绍了 Flume 的基本架构，但是通过对这些核心组件进行各种设置，可以根据需求来构建不同的拓扑。一些众所周知的结构是：

- 分布式数据流水线
- 扇出
- 扇入

下一小节将详细描述这些结构。此外，还有另外一节详细介绍了高级的 Flume 架构，这个高级架构会引入更多的组件到基本组件中。

6.4.1 Flume 架构之一：分布式数据流水线

如图 6-7 所示，各个 Source、Channel 和 Sink 单元可以一个接一个地链接起来，组成分布式数据流水线的架构。

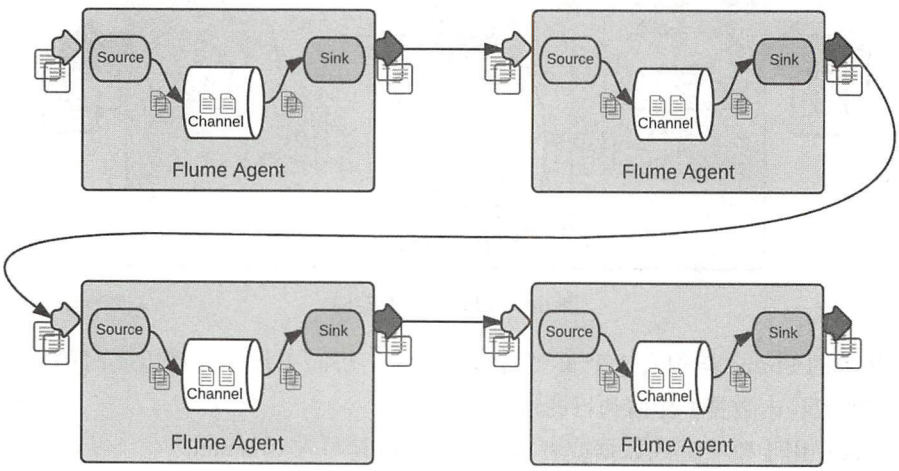


图 6-7 Flume 的分布式流水线架构

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

6.4.2　Flume 架构之二：扇出

Flume 的组件（即 Source 和 Channel）以如下方式部署：一个 Source 将数据传入多个 Channel，这种架构在术语上叫作扇出架构。图 6-8 中显示了扇出架构。

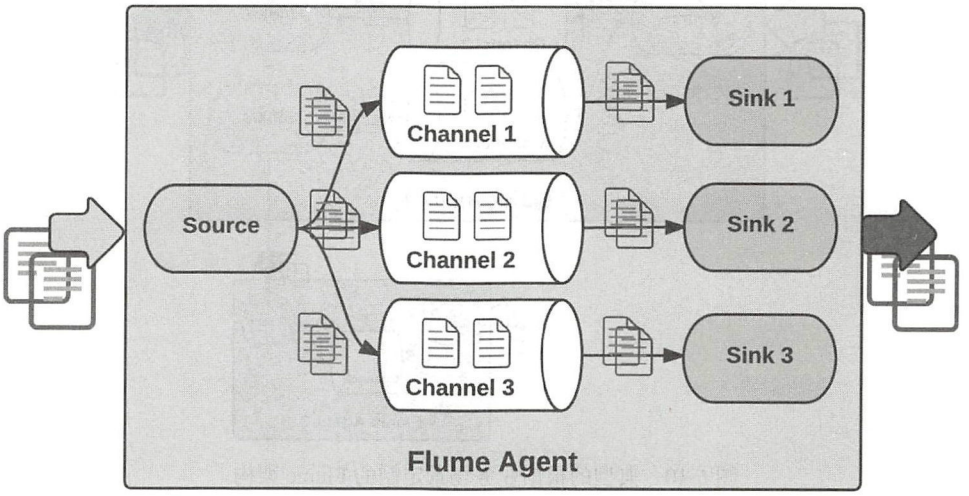


图 6-8　Flume 的扇出架构

6.4.3　Flume 架构之三：扇入

Flume 的组件（即 Source 和 Channel）以如下方式部署：多个 Source 将数据传入一个 Channel，这种架构在术语上叫作扇入架构。图 6-9 显示了扇入架构。

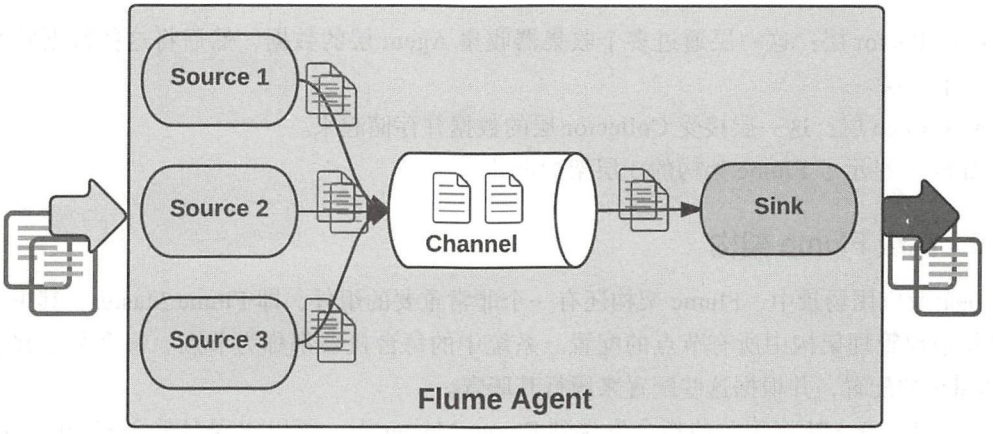


图 6-9　Flume 的扇入架构

图 6-10 中显示了这些 Flume 组件能够以更复杂地的方式组合在一起，更适合某些特定案例的功能需求。这也说明了读者可以根据自身的应用场景，自行组合使用这些组件。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

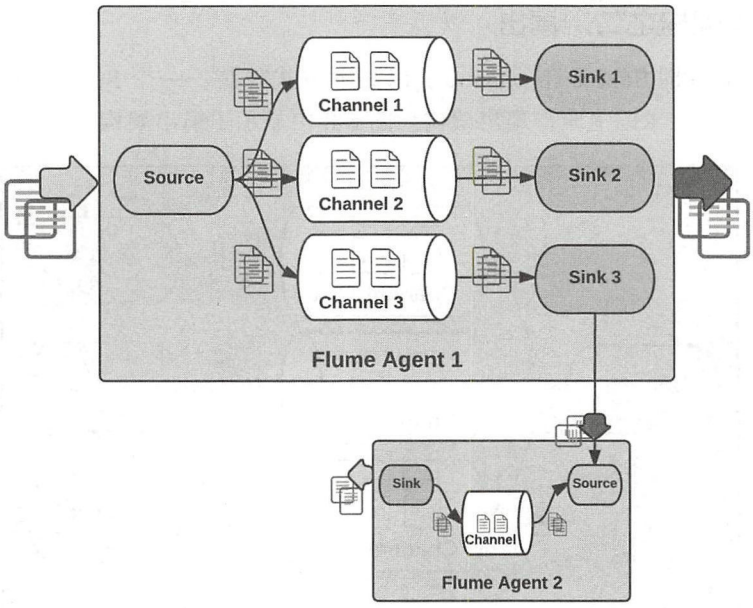


图 6-10 典型的根据业务场景定制的 Flume 架构

6.4.4 Flume 架构中的 3 层设计

大多数情况下，部署 Flume 都遵循了 3 层架构设计，这种设计也有完善的文档支持。这种设计中的 3 个主要层次如下。

- Agent 层：这一层包含了 Flume 的 Agent 组件，与需要传输数据的数据源连接在一起。
- Collector 层：这一层通过多个收集器收集 Agent 层的数据，然后将这些数据转发到下一层。
- Storage 层：这一层接受 Collector 层的数据并存储起来。

图 6-11 显示了 Flume 架构的 3 层架构设计。

6.4.5 高级 Flume 架构

在企业应用场景中，Flume 架构还有一个非常重要的组件，即 Flume Master，其主要作用是集中地管理架构中所有节点的配置。系统中的每台机器都称为节点。每个节点都依赖于 Master 的配置，并根据这些配置来履行其职责。

通过 Thrift API 将相关的指令发送到 Flume Master 上，可以实现对整个 Flume 集群的动态配置管理。

图 6-12 中显示了高级的 Flume 架构，其中 Flume Master 可以控制和支配整个集群的工作和拓扑结构。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

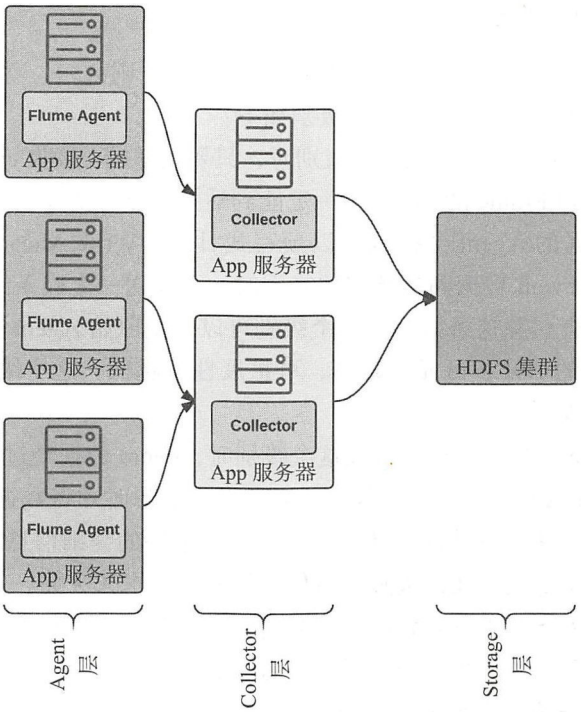


图 6-11　Flume 的 3 层架构设计

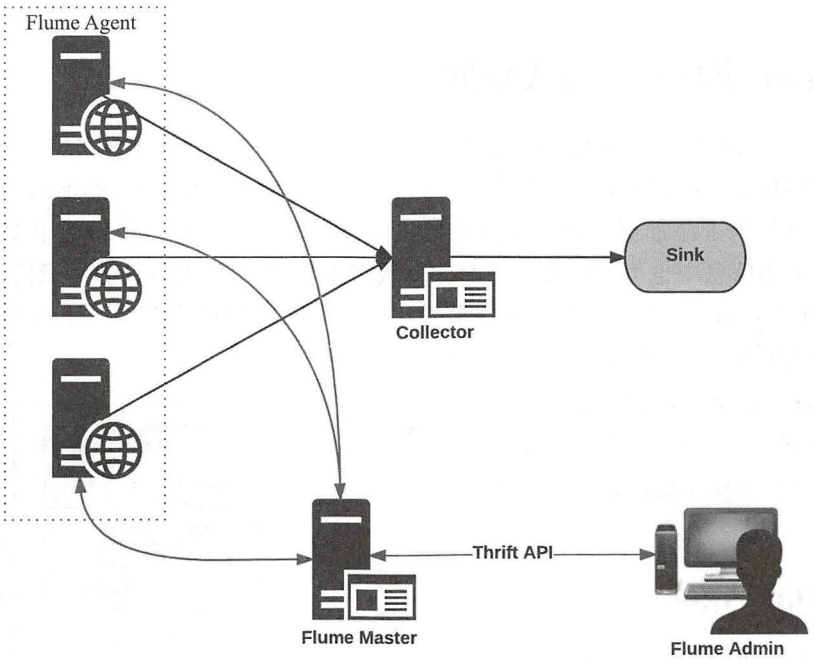


图 6-12　高级 Flume 架构

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

6.4.6 Flume 的可靠性级别

在设计 Flume 架构时，可靠性是需要遵循的核心原则之一。为了实现这种可靠性，Flume 允许用户配置可靠性级别。具体的级别如下：

- End-to-end：当可靠性设置成这个级别时，只要发送端（接收 event 的 Agent）是活跃的状态，发送到 Flume 的 event 就一定能到达另一端。为了实现这个级别的可靠性，接收 event 数据的 Agent 会把数据以预写式日志（Write Ahead Log，WAL）的方式写入磁盘。当 event 数据到达预定的终端时，会发送一个收条给到始发的 Agent，然后该 Agent 才会移除这条数据。这个级别可以承受原始 Agent 之后的任何组件故障。正如前面指出的那样，可靠性越高，可扩展性就越低，这个级别是 Flume 能提供的最高的可靠性级别。
- Store on failure：当可靠性设置成这个级别时，event 数据经过不同的 Agent（跳转），event 的发送方 Agent 只会在接收方 Agent 发生故障时才将数据保存到磁盘。只有在收不到来自下一跳的确认时，才会向磁盘写入事件的详细信息。这个级别更加实用，但是如果发生了毫无征兆的失败，事件数据有可能永久丢失。
- Best-effort：这个级别的可靠性是最弱的，也是最轻量级的，event 不经写入磁盘而直接被转发给下一跳，也不依赖下一个 Agent 的收条。

读者可以根据自身的应用需求来选择正确的可靠性级别，并请始终谨记，可靠性越高，可扩展性就越差，并且维护成本也越高。

6.5 Flume 事件——流式数据

数据以事件为单位在 Flume 的数据流水线中流动。事件的结构非常简单，分为两部分：

- Event Header：这部分是一组键 / 值对，以 Map<String,String> 的形式存储。这些头部信息是为了添加更多关于事件的数据。例如，可以保存该事件的重要性和优先级等。头部信息中也可以包含一个 UUID 或者事件 ID，作为事件的标识信息。
- Event Payload：这部分是一个字节数组，以 byte[] 的形式存储。主体信息的默认大小为 32KB，超过这个大小后就被截断，但在 Flume 中这个值可以配置。

图 6-13 显示了 Flume 事件的内部结构，它在 Flume 中从一个 Agent 跳转到另一个 Agent。

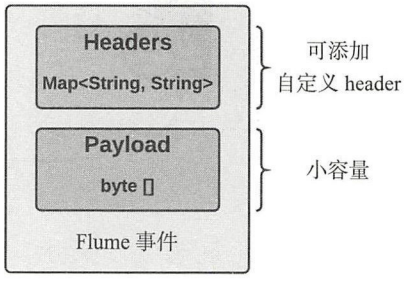


图 6-13 Flume 事件剖析

6.6 Flume Agent

Flume 的 Agent 是最小的可部署单位，以 Source、Channel 和 Sink 为主要组件。图 6-14 显示了典型的

仅供非商业用途或交流学习使用

Flume Agent 的部署情况。

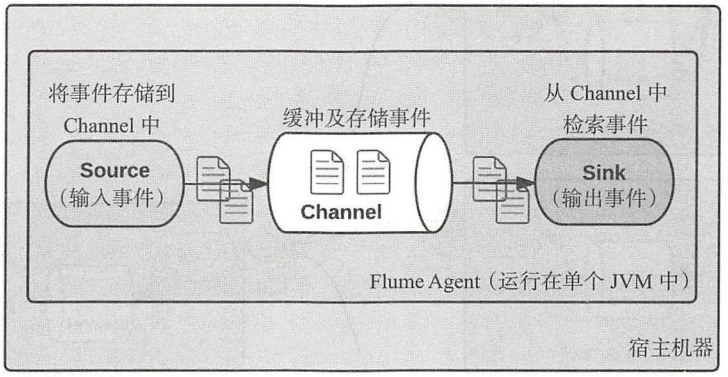


图 6-14　Flume Agent 的组件集

Flume Agent 是一个 Java 的守护进程，它接收来自数据源的事件并传递到 Channel，Channel 通常会把数据写入磁盘（根据设置的可靠性级别），然后将事件发送到 Sink。当 Sink 接收到事件时，它将回发确认信息给 Channel，然后 Channel 从存储中移除该事件。Agent 占用的内存非常小（-Xmx20m），而且能够以声明式配置进行控制。

Flume Agent 的配置

在 Flume 架构的章节中，我们已经详细讨论过关于配置的部分内容，但是我们仍然认为有必要把 Agent 的配置部分单列出来讨论。为了避免重复，部分已经讲述过的内容会直接引用进来。

以下是可以设置 Agent 的主要配置：

- 多跳式（多 Agent 数据流）：与前文中 Flume 架构章节所述的分布式数据流水线类似。一个 Agent 将事件送到另一个 Agent(称为一跳)，并根据应用需求进行配置。图 6-15 显示了多跳式 Flume Agent 的配置方式。

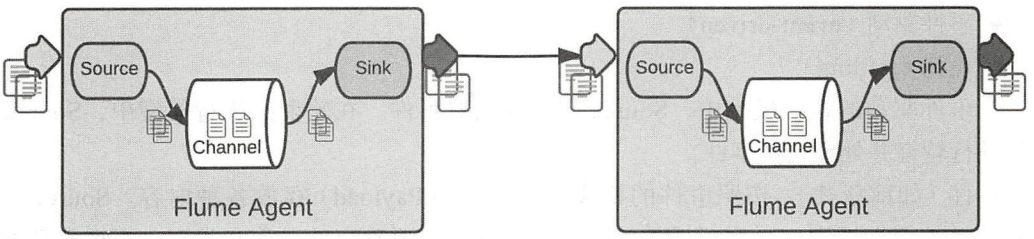


图 6-15　多跳式 Flume Agent 的配置

- 合并式（融合的数据流）：与前文中 Flume 架构章节所述的 3 层设计类似。多个 Agent 把事件发送到同一个 Agent 来完成数据流的融合。图 6-16 显示了 Agent 的实际配置。

仅供非商业用途或交流学习使用

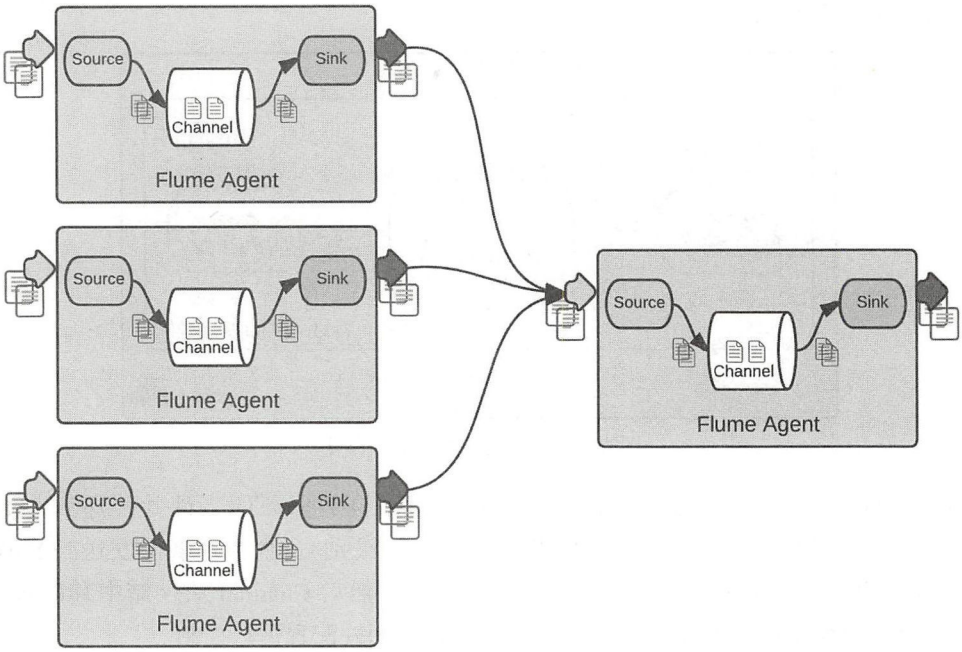


图 6-16 合并式 Flume Agent 配置

- 倍增式：与图 6-10 类似，在一个 Agent 内部，一个 Source 向多个 Channel 扇出，其中一个 Channel 向另一个 Source 发送数据。根据用例的需求，可以采用各式各样的设置。

6.7 Flume Source

Flume Agent 可以有多个 Source，但必须至少有一个 Source 才能工作。Source 由 Source Runner 管理，它控制线程方面和执行模型，如下所示：

- 事件驱动（event-driven）
- 轮询（polling）

在事件驱动的执行模型中，Source 监听并消费事件。在轮询的执行模型中，Source 对事件进行轮询并处理这些事件。

事件（如前所述），按照事件的格式（Header 和 Payload）存放各种内容。Source 遵循架构的可扩展性原则，采用了插件的方式。Source 需要确定的名称和类型。根据类型的不同，Source 需要额外的参数，而且必须进行相应的设置来保证正常工作。Source 可以接受单条事件或者批量事件（在理想情况下，是微批量而不是常规级别的批量）。Flume 中内置的 Source 可以大致划分如下：

1）异步 Source：发送事件的客户端不需要处理传输失败的情况。事件一经发送就被遗

仅供非商业用途或交流学习使用

忘。一些例子如下：

- Exec 通过命令行获取数据。配置如下：

```
agent.sources.http.type=http
agent.sources.http.port-8080
```

- Syslog（spooling directory，假脱机目录）解析文件并摄取数据。下面的配置就是配置一个假脱机目录的例子：

```
agent.sources.spool.type = customerDataDir
agent.sources.spool.spoolDir = /data/lake/customerdata
agent.sources.spool.deletePolicy = immediate
```

2）同步 Source：事件发送后，如果 Source 不向客户端确认，客户端可以正常处理这个故障场景。一些例子如下：

- JMS，通过 Java Messaging Service（Queue 和 Topic）生成和处理事件。连接到 AMQP 队列的样例配置如下：

```
agent.sources.jms.type = jms
agent.sources.jms.initialContextFactory =
org.apache.activemq.jndi.ActiveMQInitialContextFactory
agent.sources.jms.providerURL =
tcp://datalakeserver:61616
agent.sources.jms.destinationName = customerData
agent.sources.jms.destinationType = customerDataQueue
```

- HTTP 启动一个内嵌的 Web 服务器并提供 REST 风格的 API。一个配置的例子如下：

```
agent.sources.execSource.type=exec
agent.sources.execSource.command='ps -ef | grep java'
```

自定义 Source

如果读者的应用需要特殊的 Source，可以通过实现 Source 接口来写一个自定义的 Source。在一个 Agent ag1 中，可以按照如下的配置来实现自定义的 Source（class com.datalakebook.CustomSource）：

```
ag1.sources = src1
ag1.channels = ch1
ag1.sources.src1.type = com.datalakebook.CustomSource
ag1.sources.src1.channels = ch1
```

6.8　Flume Channel

Channel 是 Flume Agent 用来将数据从 Source 传输到 Sink 的一种机制。这些事件存储在 Channel 中，在事件被传送或者被 Sink 取走之前，它们将一直驻留在 Channel 中。如果 Channel 中的数据持久化存储，当 Sink 发生故障时，数据会被保存到实际的存储（HDFS）

仅供非商业用途或交流学习使用

中，并允许 Sink 重新处理这些事件。

Channel 可以粗略分为如下两类：

1. In-memory（内存）

在 Channel 组件处于活动状态时，事件都是可用的：

- 队列：这种 Channel 其实是一种内存中的队列。因为事件是存储在内存中的，所以这种 Channel 拥有极低的时延。

2. Durable（持久化）

即使 Channel 组件宕机，事件的存储仍然是可用的，在 Channel 恢复后，会继续处理这些事件：

- 文件（如 Write-Ahead Log，WAL）：这是 Channel 最常用的类型。这种 Channel 比较耐用，但需要磁盘是 RAID、SAN 或者其他类似的情形。
- JDBC：需要一个合适的 RDBMS 在后台支持这个 Channel，提供了对 ACID 的支持。
- Kafka：存储在 Kafka 集群中。

另外，还有一个特殊的 Channel 叫作 Spillable Memory Channel，它将数据存储在内存和磁盘上。当内存的空间被用尽时，剩下的事件会存储到磁盘上（内置的文件 Channel）。

读者应该还记得在本章的前面部分讨论了可靠性。如前文所述，可靠性方面取决于 Channel 类型的配置。Channel 也需要保障事件的顺序，同时保证 Agent 内部的事务。

RAID（原先是指用廉价磁盘组建的冗余阵列，现在通常是独立磁盘组建的冗余阵列）是一种数据存储的虚拟化技术，它将多个物理磁盘驱动器整合到一个逻辑单元中，以实现数据冗余或者提高性能，或者两者兼而有之。

存储区域网络（SAN）是提供对整合的、块级别的数据存储的访问的网络。SAN 主要用于增强服务器可以访问的存储设备，如磁盘阵列、磁带库和光盘库，以便这些存储像本地连接的设备一样呈现给操作系统。

在计算机科学领域，预写式日志（WAL）是在数据库系统中提供原子性和持久性（ACID 中的两个属性）的一系列技术。在使用 WAL 的系统中，所有修改操作都会在生效之前写入日志。通常重做和撤销的信息都记录在日志中。

在计算机科学中，ACID（原子性，一致性，隔离性，持久性）指的是数据库事务的特征。在数据库的概念中，对数据的单一逻辑操作被称为事务。

——维基百科

自定义 Channel

用户可以利用 Flume 的可插拔性编写自定义 Channel 来满足应用的需求。为此，必须开发一个类实现 Channel 接口。下面是一个为 Agent ag1 开发的一个自定义 Channel 的配置

仅供非商业用途或交流学习使用

范例，这个 Channel 类是 com.datalakebook.CustomChannel：

```
ag1.channels = ch1
ag1.channels.ch1.type = com.datalakebook.CustomChannel
```

6.9 Flume Sink

与 Source 的情况类似，Sink 由 Sink Runner 管理，Sink Runner 管理线程和执行模型。然而，与 Source 不同的是，Sink 的执行模式是对事件进行轮询。Sink 组件将事件（根据所需的输出类型）输出到系统外部或者其他的 Source 组件。Sink 组件也参与事务管理，当 Sink 的输出成功时，会向 Channel 发送一个确认信息，然后 Channel 将事件从持久性机制中移除。事务管理将在一个专门的小节中详细进行介绍。

目前有多种现成的 Sink 可用，具体如下：

- HDFS：数据写入 HDFS。目前支持写入文本和二进制文件（可压缩）。以下是针对名为 a1 的 Agent 的 HDFS Sink 配置示例（摘自 Flume 用户指南）。完整的配置可以在 Flume 用户指南（https://flume.apache.org）中找到。

```
a1.channels = c1
a1.sinks = k1
a1.sinks.k1.type = hdfs
a1.sinks.k1.channel = c1
a1.sinks.k1.hdfs.path = /flume/events/%y-%m-%d/%H%M/%S
a1.sinks.k1.hdfs.filePrefix = events-
a1.sinks.k1.hdfs.round = true
a1.sinks.k1.hdfs.roundValue = 10
a1.sinks.k1.hdfs.roundUnit = minute
```

- HBase：数据写入 HBase。
- AsyncHBase：数据异步写入 HBase。
- Hive：写入文本或者 JSON 到 Hive 的表和分区中。
- Null 和 Looger：用于程序调试。
- Kafka：将数据发布到 Kafka 中。本书的用例将使用这种 Sink。
- 其他类型。

因为本书没有涵盖 Sink 的所有方面，读者可以参阅 Flume 用户指南（https://goo.gl/U8pS35），其中有关于 Flume Sink 及其配置的详细信息列表。

自定义 Sink

可以利用 Sink 的可插拔特性编写自定义 Sink。读者需要根据自身的需求，实现 Sink 接口来开发自定义 Sink。下面是一个配置的样例，com.datalakebook.CustomSink 类实现了 Sink 接口，并将这个 Sink 类配置给名为 ag1 的 Agent：

仅供非商业用途或交流学习使用

```
ag1.channels = ch1
ag1.sinks = cus1
ag1.sinks.cus1.type = com.datalakebook.CustomSink
ag1.sinks.cus1.channel = ch1
```

6.10 Flume 配置

用户可以使用 Flume 的配置文件对 Flume 进行全方位的配置。一图胜千言，图 6-17 可以解释 Flume 的配置。本书并不打算覆盖 Flume 配置的所有方面，但是会解释其中最核心的部分，这也可以为理解完整的配置打下基础。

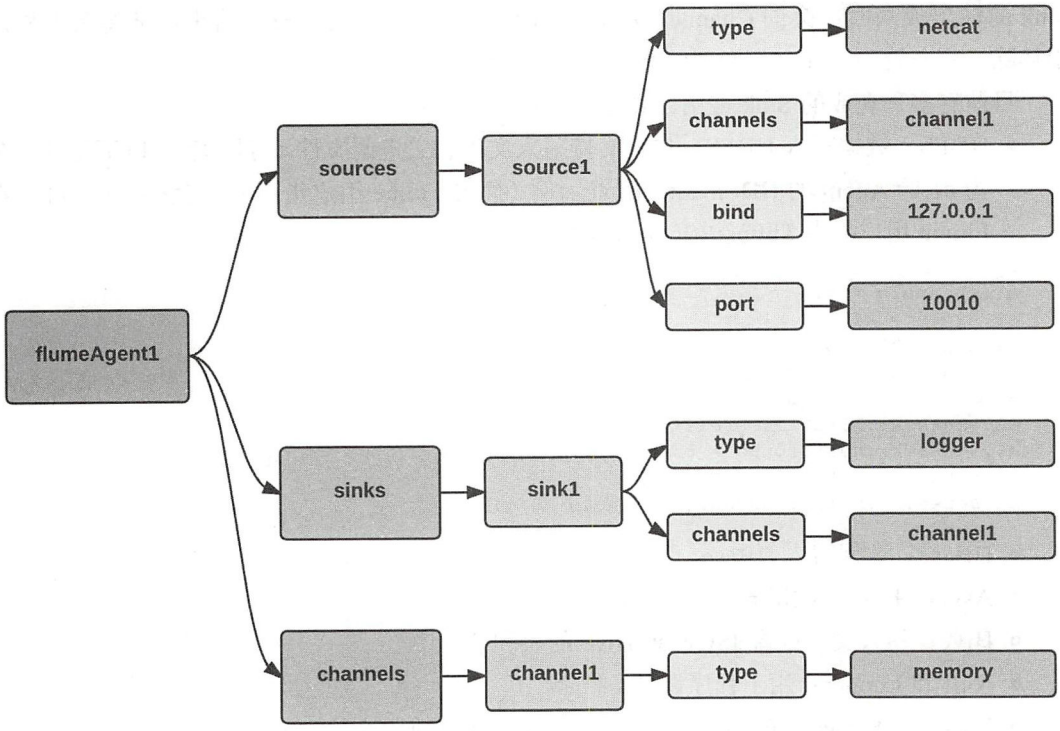


图 6-17 Flume 配置树（范例）

下面的代码对应前面树状图的配置信息：

```
# Active Flume Components
flumeAgent1.sources=source1
flumeAgent1.channels=channel1
flumeAgent1.sinks=sink1

# Define and Configure Source 1
flumeAgent1.sources.source1.type=netcat
flumeAgent1.sources.source1.channels=channel1
flumeAgent1.sources.source1.bind=127.0.0.1
```

```
flumeAgent1.sources.source1.port=10010

# Define and Configure Sink 1
flumeAgent1.sinks.sink1.type=logger
flumeAgent1.sinks.sink1.channels=channel1

# Define and Configure Channel 1
flumeAgent1.channels.channel1.type=memory

# Other Flume Agent configurations
flumeAgent2.sources=source2
…
```

Flume 在实际运行中加载配置，需要遵循一些定义好的规则。部分规则如下：

- 每个 Agent 至少包含一个 Channel。
- 每个 Source 至少包含一个 Channel。
- 每一个 Sink 有且仅有一个 Channel。
- 应该给 Agent 命名，只有已命名的 Agent 才会被加载。
- 在已命名的 Agent 中，只有活跃的组件会被加载。
- 定义任何组件时都应该定义它的类型。

6.11　Flume 事务管理

在前面的章节中，我们在不同阶段的都谈及了事务方面的内容。图 6-18 以更加直观的方式囊括了这些内容。

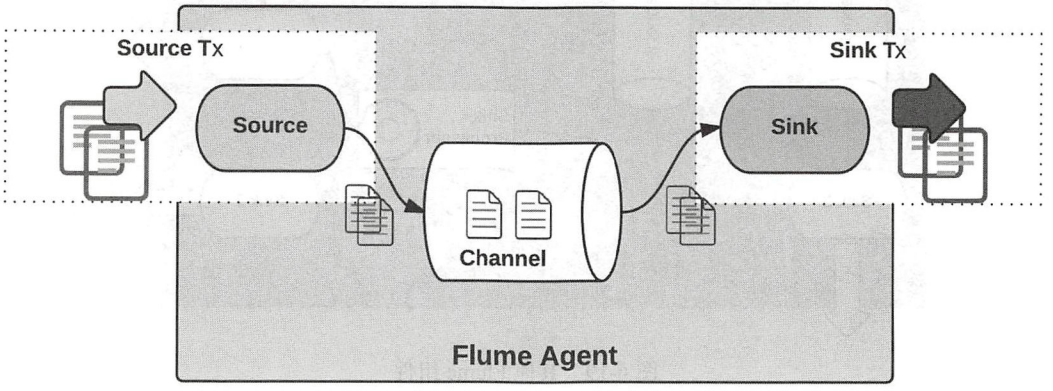


图 6-18　Flume 中的事务管理（Source Tx 和 Sink Tx）

如图 6-18 所示，数据从客户端或者上一个 Sink 进入当前的 Agent 会触发第一个事务，这种事务在图中称为 Source Tx。当事件在 Channel 中持久化并收到确认信息后，Source Tx 即宣告结束。

在 Agent 的范围内，第二个事务被称为 Sink Tx，该事务开始于 Sink 轮询数据，并当数

据向下游传输成功后，Channel 会利用确认信息来移除数据。

Flume 的事务管理涉及了方方面面的内容，根据应用的需求，可以在 Channel 中采用不同的可靠性级别来决定如何实现事务（Sink Tx）。

6.12 Flume 的其他组件

除了前面提到的这些主要组件之外，Flume 还有其他一些非常重要的组件。本节将详细介绍这些组件。图 6-19 展示了这些组件是如何协作的。

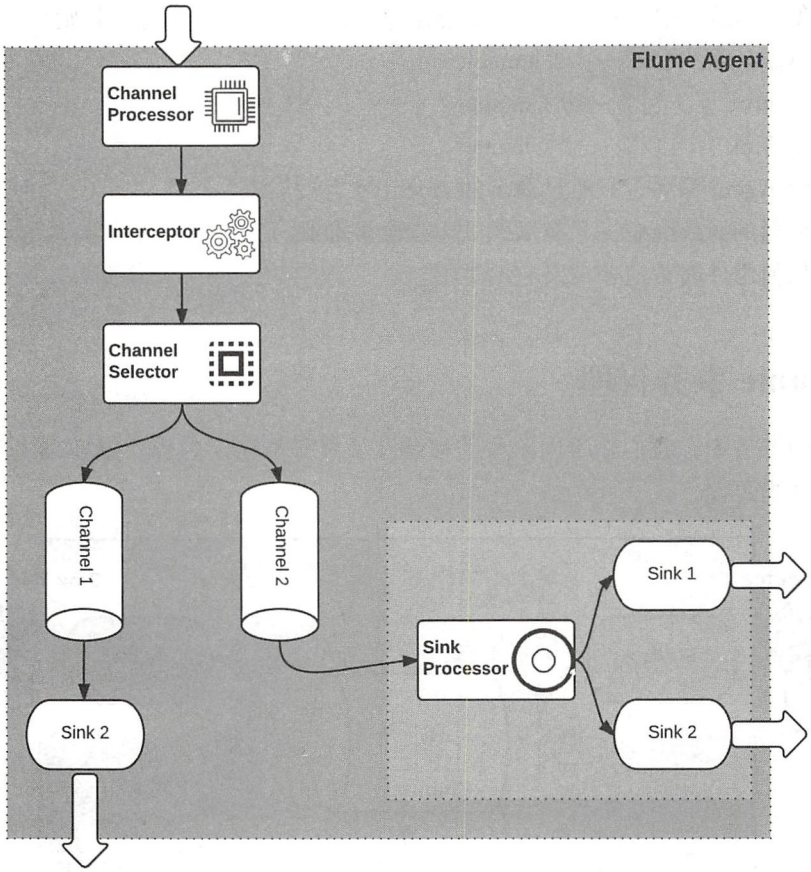


图 6-19 其他 Flume 组件

下一小节将深入探讨图 6-19 中这些组件的功能和作用。了解这些组件，将有助于读者设计正确的组件拓扑，以便保证应用的成功。

6.12.1 Channel Processor

如图 6-19 所示，Source 将事件发送到 Channel Processor。每个 Source 都有自己的

Channel Processor，为了将 Channel 中的事件保存起来，Source 将工作委托给 Channel Processor，它会根据 Channel 的实际类型进行数据持久化。

6.12.2　Interceptor

如图 6-19 所示，Channel Processor 随后将事件传输给 Interceptor。Channel Selector 将实例化一个 Interceptor 来使用。

Interceptor 作用于事件产生之后发送到 Channel 之前这段时间。得益于 Interceptor 机制，Flume 可以对事件进行修改和过滤。Interceptor 主要分为如下两类：

1. 内置 / 预定义的 Interceptor

顾名思义，这些 Interceptor 是 Flume 内置的。例如：

- Timestamp Interceptor：向事件的 header 域写入毫秒数格式的时间戳。
- Host Interceptor：在 Agent 运行时写入机器的 host 或者 IP 信息。
- UUID Interceptor：给每个事件中加入一个 UUID 作为标记。
- Morphline Interceptor：通过预定义一个 Morphline 配置文件，将这些事件用文件中的命令执行一遍，对事件进行过滤或者转换。
- Search and Replace Interceptor：提供基于正则表达式的搜索和替换等功能。

2. 自定义 Interceptor

通过查看事件，可以增加事件 header 信息或者对事件进行转换。这种自定义的 Interceptor 是由纯 Java 代码编写的。

图 6-20 展示了 Interceptor 链的范例。

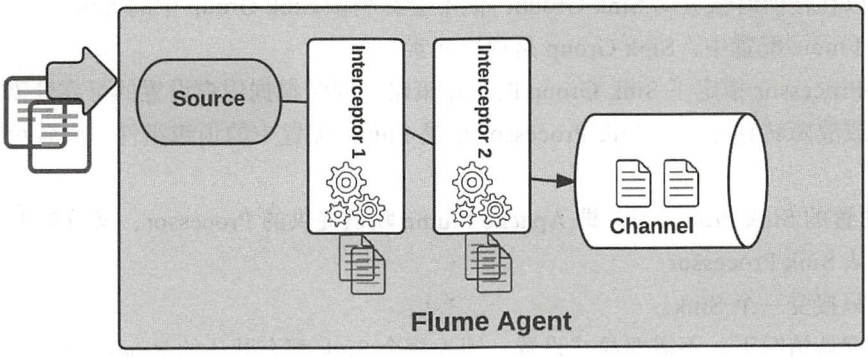


图 6-20　Interceptor 链的范例

6.12.3　Channel Selector

Channel Selector 有助于根据设置标准选择正确的 Channel。与 Interceptor 十分类似，Channel Selector 也主要分为两个类别：

1. 内置的 Channel Selector

- Replicating Channel Selector
  - 默认的 Channel Selector，如果没有显示指定类别，即采用这种 Selector。
  - 如果为 Source 指定多个 Channel，则每个 Channel 都获得该事件的副本。
- Multiplexing Channel Selector
  - 根据 header 选择 Channel。
  - 可以认为这种 Selector 提供了基于 Header 数据的动态路由。
  - 上下文路由。

2. 自定义 Channel Selector

读者可以为 Channel Selector 类编写自定义的实现，来处理动态标准。

6.12.4 Sink Group

可靠性是构建 Flume 的核心架构原则之一。为了对应各种可能的故障，Flume 的组件可以按多种方式搭配组合。如果在 Channel 传输数据到 Sink 时出现故障，Sink 可以设置成负载均衡或者故障转移的方式。要实现这种方式，一个先决条件是配置一个 Sink Group。顾名思义，Sink Group 是逻辑上的一个 Sink 群组。这些群组负责实现负载均衡和故障转移。

Sink Group 的一些特性值得注意，如下所列：

- 一个 Sink 一次只能与一个 Sink Group 关联。
- 所有的 Sink 属于自己默认的 Sink Group。
- 如果没有特殊指定，每个 Sink 都属于默认的 Sink Group，由它来负责事件的传输。
- 可以在任何时候关闭 Sink Group，但不会影响该 Sink Group 中的 Sink。
- 在 Flume 配置中，Sink Group 是顶层元素。

Sink Processor 指定了 Sink Group 的工作策略，同时根据用户设置的可靠性需求来实现负载均衡或故障转移方案。Sink Processor 也是 Flume 配置中的顶级组件。大致可以分为如下两类：

1）内置的 Sink Processor。即 Apache Flume 默认提供的 Processor，又分如下 3 类：

- 默认 Sink Processor
  - 只接受一个 Sink。
  - 默认情况下，不需要显式设置，因为每个 Sink 都有默认的 Processor。
- 故障转移 Sink Processor
  - 拥有一个 Sink 的优先级列表。
  - 使用该优先级来选择 Sink，并确保总有一个 Sink 来处理事件。
  - 如果事件发送到某个 Sink 失败，则下一个事件会自动选择优先级列表中的下一个 Sink。

- 负载均衡 Sink Processor
  - 拥有一个带索引的 Sink 列表。
  - 当事件到来时，它使用传统的负载均衡方法来查看负载，并尽可能均匀地分发负载。
  - 默认的策略是轮询式负载平衡。
  - 另一个策略是随机负载平衡。

2）自定义的 Sink Processor。已经有这方面的支持，但尚未在最新的 Flume 版本中集成。

6.12.5　事件序列化

为了将事件转化为用户需要的格式，在安装 Flume 时就集成了一些序列化组件。目前只有少数 Sink 支持序列化器（serializer），即 File_Roll Sink 和 HDFS Sink。Flume 的默认安装包中序列化工具很少，具体如下：

1. 正文文本序列化器（body text serializer）

- 将事件的主体数据写入输出流。
- 进行流输出时舍弃事件的 header 信息。

2. Flume 事件 –Avro 事件序列化器（Flume event-Avro event serializer）

Flume 事件 –Avro 事件序列化器可将事件写入 Avro 容器文件。

3. Avro 事件序列化器（Avro event serializer）

Avro 事件序列化器与前面介绍的序列化工具类似，但是可以控制 Avro 容器文件的记录模式。

6.13　上下文路由

如前文所述，事件有两个主要部分，即 Header 和 Payload。Header（键 / 值对）中的值可以用来定义路由。可以决定路由选择的两个组件如下：

- Channel：可以根据 Header 的值选择 Channel，还可以通过一个自定义的组件（即 Channel Selector）来选择 Channel。用户可以根据业务需求写代码来实现 Channel 的选择。
- Sink：跟前面一样，Header 中的值可以用来选择正确的 Sink。同样地，对 Sink 也可以通过自定义来实现不同的操作，以满足用户的需求。Header 中一些默认的值也可以用来处理复杂问题，进而满足用户的需求。

基本上，用户可以引入任意数量的 Header 数据，使得组件可以正确地运行。这样一来，Flume 的各个组件就可以全面地动态运行了。

仅供非商业用途或交流学习使用

6.14 Flume 运行范例

在本节中，我们将一如既往地提供该技术的完整用例；在本节的最后，有一个专门的小节介绍如何在 SCV 案例中应用该技术，包括一些真实的代码片段。

6.14.1 安装和配置

该步骤详细介绍了绝大部分安装工作，这些工作是保障 Flume 正常工作的必要步骤，也是应用 Flume 的先决条件。

第 1 步：安装和验证 Flume

在本节中，我们将安装 Apache Flume，然后验证安装是否成功。按照如下的给定步骤即可完成安装：

1）使用以下命令下载 Apache Flume 的二进制发行版，我们将使用当前版本的 Apache Flume，也就是 1.7.0。

```
wget http://www-us.apache.org/dist/flume/1.7.0/apache-flume-1.7.0-b
in.tar.gz
```

2）完成下载后，通过以下命令将 Flume 解压到用户打算安装的目录：

```
tar -zxvf ${DOWNLOAD_DIRECTORY}/apache-flume-1.7.0-bin.tar.gz
```

3）我们用 ${FLUME_HOME} 来表示 Flume 的安装目录。将 ${FLUME_HOME} 设置为环境变量，在 ~/ .bashrc 文件中追加以下命令，其中应将 ${FLUME_HOME} 替换为所需的完整路径：

```
export FLUME_HOME=${FLUME_HOME}
```

4）解压安装完成后，只需要简单地在 conf/flume-conf.properties 文件中加入所需的配置就能使用 Flume 了。

第 2 步：配置 Flume

Flume 的配置文件夹有 conf 文件夹，其中包含各种配置文件的模板。这些模板有完善的文档说明，能帮助用户快速上手。查看 conf/flume-conf-template.properties，可以看到如图 6-21 所示的内容。

在之前介绍 Flume 配置的部分中，已经涵盖了配置各个方面的细节内容，此处不再赘述。在下一个章节中将介绍如何配置 Flume 来适配 SCV 用例的需求。

第 3 步：启动 Flume

可以使用如下的 flume-ng 命令启动 flume（样例仅供参考）。这些将在后面的内容中再次详细说明。

```
bin/flume-ng agent -n $agent_name conf -f conf/flume-conf.properties
```

仅供非商业用途或交流学习使用

```
# The configuration file needs to define the sources,
# the channels and the sinks.
# Sources, channels and sinks are defined per agent,
# in this case called 'agent'

agent.sources = seqGenSrc
agent.channels = memoryChannel
agent.sinks = loggerSink

# For each one of the sources, the type is defined
agent.sources.seqGenSrc.type = seq

# The channel can be defined as follows.
agent.sources.seqGenSrc.channels = memoryChannel

# Each sink's type must be defined
agent.sinks.loggerSink.type = logger

#Specify the channel the sink should use
agent.sinks.loggerSink.channel = memoryChannel

# Each channel's type is defined.
agent.channels.memoryChannel.type = memory

# Other config values specific to each type of channel(sink or source)
# can be defined as well
# In this case, it specifies the capacity of the memory channel
agent.channels.memoryChannel.capacity = 100
```

图 6-21　Flume 配置模板

此处：

agent 表示 Flume Agent 的实例。

-n $agent_name 表示配置文件中定义的 Agent 的名称。

-f conf/flume.conf.properties 表示定义 Flume 各组件的配置文件。

6.14.2　SCV 用例中的 Flume

正如在前面的章节中那样，结合 SCV 用例对数据湖整体架构进行了讨论，可以在多种应用中以不同方式来使用 Flume 的功能。Flume 既可以用于数据获取，也可以在数据摄取方面发挥作用。在本章中，我们将重点讨论 SCV 用例的各种数据源和 Flume 的数据获取能力。

虽然后面的章节中会更详细地讨论消息层，为了完整地介绍 Flume 的数据获取机制，我们还将在示例中对 Apache Kafka 这个消息引擎做一些基本的介绍。

安装 Kafka

我们将运行一个基本的、最小安装的 Kafka，仅仅把它作为一个消息的分发节点（接收消息的组件），并作为一个主要的 Flume Sink。现在用默认配置先运行起来。

1）使用以下命令下载 Kafka 的安装包：

```
wget http://redrockdigimark.com/apachemirror/kafka/0.10.1.1/kafka_2
.11-0.10.1.1.tgz
```

仅供非商业用途或交流学习使用

2）切换到用户目录，使用以下命令提取 Kafka tar 压缩包中的内容：

```
tar -xzvf <DOWNLOAD_DIRECTORY>/kafka_2.11-0.10.1.1.tgz
```

3）将 Kafka 文件夹解压缩到 ${KAFKA_HOME}，并使用以下命令来配置，同时将其添加到 ~/.bashrc 文件中。另外，与安装其他组件一样，用户可以选择将 ${KAFKA_HOME}/bin 添加到 $PATH 中：

```
export KAFKA_HOME=${KAFKA_HOME}
```

4）切换到 Kafka 的安装目录 ${KAFKA_HOME}，然后运行以下命令来启动 Kafka 的服务端：

```
${KAFKA_HOME}/bin/zookeeper-server-start.sh
${KAFKA_HOME}/config/zookeeper.properties
```

然后在另一个 bash shell 中，运行以下命令启动 Kafka 服务端：

```
${KAFKA_HOME}/bin/kafka-server-start.sh
${KAFKA_HOME}/config/server.properties
```

5）成功启动 Kafka 服务端后，用户可以在 shell 控制台上看到已启动信息（kafka.server.KafkaServer）。

6）该 Kafka 实例默认依赖的配置文件是 server.properties，该文件打包在 Kafka 的二进制文件中。为了支持 Flume 来实现数据获取，这已经完全够用了。

在下面的章节中，我们将详细介绍 SCV 场景中需要的一些例子。一种情况是从数据库加载数据，而另一种情况是从非结构化的数据源（如假脱机文件（spool file））加载数据。我们将看到如何运用这些例子来实现这些场景。

案例一：从 RDBMS 到 Kafka

由于在前面的章节中，已经构造了一些数据用于实践 Sqoop 的数据导入 / 导出，本案例中将复用这些数据，用 Flume 将这些数据作为事件流式地写入 Kafka。数据库配置过程如下：

1）复制在第 5 章下载的 PostgresSQL 驱动到 ${FLUME_HOME}/lib 目录下：

```
cp ${SQOOP_HOME}/lib/postgresql-9.4.1212.jre6.jar
$F{LUME_HOME|/lib
```

2）SQL 不属于 Flume 的标准 Source，因此，需要下载并安装第三方的 Source：

①从下述地址下载这个 Source，命令如下：

```
wget https://github.com/keedio/flume-ng-sql-source/archive/
1.4.2.tar.gz
```

②将文件重命名为 sql-source-1.4.2.tar.gz：

```
mv ${DOWNLOAD_DIR}/1.4.2.tar.gz
${DOWNLOAD_DIR}/sql-source-1.4.2.tar.gz
```

仅供非商业用途或交流学习使用

③用下述命令解压 tar 包：

```
tar -zxvf ${DOWNLOAD_DIRECTORY}/sql-source-1.4.2.tar.gz
```

④切换到解压目录并用以下命令编译源代码。二进制 JAR 包将被编译到目标文件夹中：

```
mvn install -DskipTests
```

⑤用下列命令创建 plugins.d 目录，并创建 sql-source 等子目录：

```
mkdir -p ${FLUME_HOME}/plugins.d/sql-source/lib
```

⑥使用下列命令从目标文件夹中复制编译好的 .jar 文件到 plugins.d/sql-source/lib 文件夹：

```
cp ${DOWNLOAD_DIR}/flume-ng-sql-source-1.4.2/target/flume-ng-
sql-source-1.4.2.jar
${FLUME_HOME}/plugins.d/sql-source/lib
```

3）将目录切换回 ${FLUME_HOME} 并创建一个新的 Flume 配置文件 db-kafka-flume-conf.properties。按照下列方式修改配置：

- Source 配置

使用下列属性来配置 Source。根据读者的虚拟机设置来修改属性。在启动 Flume Agent 之前，需要保证 agent.sources.sqlsource.status.file.path 属性配置的目录已经存在，在下列配置中，这个目录是 mkdir~/db-kafka。在 Flume 的配置文件中，需要指定所有目录和文件的完整路径：

```
agent.sources = sql-source
agent.sources.sql-
source.type=org.keedio.flume.source.SQLSource
agent.sources.sql-
source.hibernate.connection.url=jdbc:postgresql://<ip_addre
ss>/sourcedb?schema=public
agent.sources.sql-source.hibernate.connection.user=postgres
agent.sources.sql-
source.hibernate.connection.password=<db_password>
agent.sources.sql-source.table=customer
agent.sources.sql-source.columns.to.select=*
agent.sources.sql-source.status.file.path=/home/centos/db-
kafka
agent.sources.sql-source.status.file.name=sql-source.status
```

前面设置的各属性说明如表 6-4 所示。

表 6-4　Flume Source 配置

属　性	说　明
agent.sources	为 Source 设置一个名称
agent.sources.sql-source.type	配置 Source 的类型。这里设置的 SQL 类 Source 属于 GitHub 上的开源项目

仅供非商业用途或交流学习使用

（续）

属　性	说　明
agent.sources.sql-source.hibernate.connection.url	设置 JDBC 的 URL
agent.sources.sql-source.hibernate.connection.user	设置连接的用户名
agent.sources.sql-source.hibernate.connection.password	设置连接的密码
agent.sources.sql-source.table	设置表名
agent.sources.sql-source.columns.to.select	设置要 select 的列。这里设置成 * 代表 select 所有的列
agent.sources.sql-source.status.file.path	设置一个位置存放监控 Source 的状态文件
agent.sources.sql-source.status.file.name	设置状态文件的名字

- Channel 配置

```
# The channel can be defined as follows.
agent.channels = memoryChannel
agent.sources.sql-source.channels = memoryChannel
agent.channels.memoryChannel.type = memory
agent.channels.memoryChannel.capacity = 100
```

- Sink 配置

```
# Each sink must be defined
agent.sinks = kafkaSink
agent.sinks.kafkaSink.type=org.apache.flume.sink.kafka.KafkaSink
agent.sinks.kafkaSink.brokerList=localhost:9092
agent.sinks.kafkaSink.topic=db
agent.sinks.kafkaSink.channel = memoryChannel
```

4）整体配置应该基本如下：

```
agent.sources = sql-source
agent.sources.sql-source.type=org.keedio.flume.source.SQLSource
agent.sources.sql-
source.hibernate.connection.url=jdbc:postgresql://<ip_address>/sour
cedb?schema=public
agent.sources.sql-source.hibernate.connection.user=postgres
agent.sources.sql-
source.hibernate.connection.password=<db_password>
agent.sources.sql-source.table=customer
agent.sources.sql-source.columns.to.select=*
agent.sources.sql-source.status.file.path=/home/centos/db-kafka
agent.sources.sql-source.status.file.name=sql-source.status

# 下面对 Channel 进行定义
agent.channels = memoryChannel
agent.sources.sql-source.channels = memoryChannel
agent.channels.memoryChannel.type = memory
agent.channels.memoryChannel.capacity = 100

# 必须为每个 Sink 进行定义
agent.sinks = kafkaSink
agent.sinks.kafkaSink.type=org.apache.flume.sink.kafka.KafkaSink
agent.sinks.kafkaSink.brokerList=localhost:9092
agent.sinks.kafkaSink.topic=db
agent.sinks.kafkaSink.channel = memoryChannel
```

仅供非商业用途或交流学习使用

5）用下列命令启动 Flume Agent：

```
${FLUME_HOME}/bin/flume-ng agent --conf ${FLUME_HOME}/conf/  -f
${FLUME_HOME}/conf/db-kafka-flume-conf.properties  -n agent -
Dflume.root.logger=INFO,console
```

6）观察：

- 状态文件中的下列语句暗示了 Source 的最近一次索引读取及元信息（参见图 6-22）：

```
cat ~/db-kafka/sql-source.stat
```

```
{"SourceName":"sql-source","URL":"jdbc:postgresql:\/\/192.168.43.28\/sourcedb?schema=public","LastIndex
":"5","ColumnsToSelect":"*","Table":"customer"}[centos@localhost apache-flume-1.7.0-bin]$
```

图 6-22　Source 最近索引读取及元数据

- 以下 Kafka 日志表明事件已经进入队列就绪（参见图 6-23）：

```
cat /tmp/kafka-logs/db-0/00000000000000000000.log
```

```
[centos@localhost apache-flume-1.7.0-bin]$ cat /tmp/kafka-logs/db-0/00000000000000000000.log
5??qS????????????"0","tomcy","john","1985-10-20"4??b????????????"1","rahul","dev","1989-08-15"7?N??????
???????!"2","pankaj","misra","1982-08-10"3??@?????????????"3","devi","lal","1990-05-06"3?`????????????"
4","john","doe","1992-06-25"[centos@localhost apache-flume-1.7.0-bin]$
```

图 6-23　Kafka 日志显示进入队列的事件

- 以下消费者命令可启动消费者程序，并显示从 Kafka 中消费的事件信息（参见图 6-24）：

```
${KAFKA_HOME}/bin/kafka-console-consumer.sh
${KAFKA_HOME}/config/consumer.properties --topic db -
bootstrap-server localhost:9092 --from-beginning
```

```
[centos@localhost kafka_2.11-0.10.1.1]$ bin/kafka-console-consumer.sh ./config/consumer.properties  --t
opic db -bootstrap-server localhost:9092 --from-beginning
[2017-02-16 07:24:37,025] INFO [GroupCoordinator 0]: Preparing to restabilize group console-consumer-47
468 with old generation 0 (kafka.coordinator.GroupCoordinator)
[2017-02-16 07:24:37,029] INFO [GroupCoordinator 0]: Stabilized group console-consumer-47468 generation
 1 (kafka.coordinator.GroupCoordinator)
[2017-02-16 07:24:37,057] INFO [GroupCoordinator 0]: Assignment received from leader for group console-
consumer-47468 for generation 1 (kafka.coordinator.GroupCoordinator)
"0","tomcy","john","1985-10-20"
"1","rahul","dev","1989-08-15"
"2","pankaj","misra","1982-08-10"
"3","devi","lal","1990-05-06"
"4","john","doe","1992-06-25"
```

图 6-24　Consumer 命令显示被 Kafka 消费的事件

案例二：从假脱机文件到 Kafka

简单来说，假脱机文件是包含待处理数据的文件。大多数情况下，这些文件包含带分隔符的信息（由字符分隔），可以逐行进行读取处理，其中每行代表一条记录。这些行文本数据也可以包含 XML/JSON 格式的数据。

由其他系统输出的假脱机文件可以作为一个 Flume Source，其中可能包含用户数据，

仅供非商业用途或交流学习使用

所以这些假脱机文件可以集成到数据湖中。

Flume 框架支持多种假脱机格式，此处我们考虑使用最常见的格式，其中包含 JSON 格式的消息数据。

可以按照下列步骤配置假脱机文件 Source：

1）另创建一份配置文件 ${FLUME_HOME}/conf/spool-kafka-flume-conf.properties，与集成 db-kafka 的方法基本一样。

2）按照下列方法配置 Source、Channel 和 Sink：

- Source 配置

首先，将假脱机文件配置为一个 Source。使用命令 mkdir ~/flume-data 创建用户目录。该目录包含需要处理的文件：

```
agent.sources = spool-source
agent.sources.spool-source.type=spooldir
agent.sources.spool-
source.spoolDir=/home/centos/flume-data
```

然后，确保该假脱机文件目录存在，同时保证 Flume 进程能够访问。

最后，在 ~/data 目录中，创建名为 spool-1 的假脱机文件，并将下面的数据写入该文件：

```
{"id":0,"firstName":"tomcy","lastName":"john","dob
":"1985-10-20"}{"id":1,"firstName":"rahul","lastNa
me":"dev","dob":"1989-08-15"}{"id":2,"firstName":"
pankaj","lastName":"misra","dob":"1982-08-10"}{"id
":3,"firstName":"devi","lastName":"lal","dob":"199
0-05-06"}{"id":4,"firstName":"john","lastName":"do
e","dob":"1992-06-25"}
```

- Channel 配置

Channel 的配置和上一个案例大体相同，只有一个小改动，就是会使用 memoryChannel。

```
agent.channels = memoryChannel
agent.sources.spool-source.channels = memoryChannel
agent.channels.memoryChannel.type = memory
agent.channels.memoryChannel.capacity = 100
```

- Sink 配置

Sink 的配置需要稍作修改，因为大部分配置保持不变，只有 Kafka 的 topic 需要变更，具体如下：

```
# 每个 Sink 必须被定义
agent.sinks = kafkaSink
agent.sinks.kafkaSink.type=org.apache.flume.sink.kafka.Kafk
aSink
agent.sinks.kafkaSink.brokerList=localhost:9092
agent.sinks.kafkaSink.topic=spooled
agent.sinks.kafkaSink.channel = memoryChannel
```

仅供非商业用途或交流学习使用

3）执行如下命令启动 Flume Agent：

```
${FLUME_HOME}/bin/flume-ng agent --conf ${FLUME_HOME}/conf/  -f
${FLUME_HOME}/conf/spool-kafka-flume-conf.properties  -n agent -
Dflume.root.logger=INFO,console
```

4）验证并观察：

- 假脱机文件被处理后会被重新命名为 spool -1.COMPLETED。
- Kafka 日志包含来自假脱机文件的数据流，如下所示（参见图 6-25）：

```
cat /tmp/kafka-logs/spooled-0/00000000000000000000.log
```

```
[centos@localhost apache-flume-1.7.0-bin]$ cat /tmp/kafka-logs/spooled-0/00000000000000
000000.log
W?T\?????????????A{"id":0,"firstName":"tomcy","lastName":"john","dob":"1985-10-20"}V?Mx
????????????@{"id":1,"firstName":"rahul","lastName":"dev","dob":"1989-08-15"}Y?????????
???????C{"id":2,"firstName":"pankaj","lastName":"misra","dob":"1982-08-10"}Uu?D{???????
??????{"id":3,"firstName":"devi","lastName":"lal","dob":"1990-05-06"}U)V??????????????{
"id":4,"firstName":"john","lastName":"doe","dob":"1992-06-25"}???????????????[centos@lo
```

图 6-25　Kafka 日志

- 以下命令启动 Kafka Consumer，并显示从 Kafka 中消费的事件信息（参见图 6-26）：

```
${KAFKA_HOME}/bin/kafka-console-consumer.sh
${KAFKA_HOME}/config/consumer.properties --topic spooled -
bootstrap-server localhost:9092 --from-beginning
```

```
[centos@localhost kafka_2.11-0.10.1.1]$ bin/kafka-console-consumer.sh ./config/consumer
.properties  --topic spooled -bootstrap-server localhost:9092 --from-beginning
[2017-02-16 09:08:09,863] INFO [GroupCoordinator 0]: Preparing to restabilize group con
sole-consumer-88710 with old generation 0 (kafka.coordinator.GroupCoordinator)
[2017-02-16 09:08:09,863] INFO [GroupCoordinator 0]: Stabilized group console-consumer-
88710 generation 1 (kafka.coordinator.GroupCoordinator)
[2017-02-16 09:08:09,870] INFO [GroupCoordinator 0]: Assignment received from leader fo
r group console-consumer-88710 for generation 1 (kafka.coordinator.GroupCoordinator)
{"id":0,"firstName":"tomcy","lastName":"john","dob":"1985-10-20"}
{"id":1,"firstName":"rahul","lastName":"dev","dob":"1989-08-15"}
{"id":2,"firstName":"pankaj","lastName":"misra","dob":"1982-08-10"}
{"id":3,"firstName":"devi","lastName":"lal","dob":"1990-05-06"}
{"id":4,"firstName":"john","lastName":"doe","dob":"1992-06-25"}
```

图 6-26　运行 Kafka Consumer

案例三：Interceptor

Flume 支持多种 Interceptor，在某些特定的场景中非常有用。如前文所述，Interceptor 作用于 Source 中的事件，可以拦截事件并根据需要添加一些细节内容。

为了更好地理解 Interceptor 的用途，我们以流入 Kafka 的假脱机事件（spooled event）为例进行说明。我们将拦截这些假脱机事件并添加两个有用的属性：

- Timestamp（时间戳）：为每个事件添加时间戳有助于维护事件的时间属性，从而可以跟踪事件的端到端时间。

仅供非商业用途或交流学习使用

- UUID（事件唯一标识）：添加一个 UUID 作为每个事件的唯一标识。由于事件是不可变的，所以通过 UUID 可以跟踪这些事件。

这需要创建一个新的配置文件，路径是 ${FLUME_HOME}/conf/spool-interceptor-kafka-flume-conf.properties，与假脱机文件的配置类似，只是在 Source 和 Sink 的配置中有些许改动，加入了 Interceptor 相关的配置，如下所示：

- Source 配置

Source 的配置至少定义了额外的 Interceptor，即时间戳和 UUID 以及它们各自的属性，具体如下：

```
agent.sources = spool-source
agent.sources.spool-source.type=spooldir
agent.sources.spool-source.spoolDir=/home/centos/flume-data
agent.sources.spool-source.interceptors=ts uuid

# Interceptor 时间戳定义
agent.sources.spool-source.interceptors.ts.type=timestamp

# Interceptor UUID 定义
agent.sources.spool-
source.interceptors.uuid.type=org.apache.flume.sink.solr.morphline.
UUIDInterceptor$Builder
agent.sources.spool-source.interceptors.uuid.headerName=eventId
```

- Channel 配置

Channel 的配置基本与案例二相同：

```
agent.channels = memoryChannel
agent.sources.spool-source.channels = memoryChannel
agent.channels.memoryChannel.type = memory
agent.channels.memoryChannel.capacity = 100
```

- Sink 配置

Sink 配置的改动如下：

```
# 每个 Sink 必须被定义
agent.sinks = kafkaSink
agent.sinks.kafkaSink.type=org.apache.flume.sink.kafka.KafkaSink
agent.sinks.kafkaSink.brokerList=localhost:9092
agent.sinks.kafkaSink.topic=spooled-intercepted
agent.sinks.kafkaSink.channel = memoryChannel
agent.sinks.kafkaSink.useFlumeEventFormat=true
```

正如 Sink 配置的改动，增加了额外的属性 useFlumeEventFormat，以便在从 Kafka 获取事件正文时也能获取 Header 信息。

为了运用并重新运行前面的 Interceptor 案例，需要执行以下操作：

1）使用下面的命令将假脱机文件重命名成未完成的状态：

```
mv ~/flume-data/spool-1.COMPLETED ~/flume-data/spool-1
```

仅供非商业用途或交流学习使用

2）将目录切换到 <FLUME_HOME>，并执行下列命令重新处理文件：

```
${FLUME_HOME}/bin/flume-ng agent --conf ${FLUME_HOME}/conf/  -f
${FLUME_HOME}/conf/spool-interceptor-kafka-flume-conf.properties  -
n agent -Dflume.root.logger=INFO,console
```

3）可以在控制台看到 Kafka 消费者的输出，如下所示（参见图 6-27）：

```
${KAFKA_HOME}/bin/kafka-console-consumer.sh
${KAFKA_HOME}/config/consumer.properties --topic spooled-
intercepted -bootstrap-server localhost:9092 --from-beginning
```

```
[centos@localhost kafka_2.11-0.10.1.1]$ bin/kafka-console-consumer.sh ./config/consumer.properties  --topic spooled -bootstrap-server
localhost:9092 --from-beginning
[2017-02-16 10:35:43,946] INFO [GroupCoordinator 0]: Preparing to restabilize group console-consumer-24139 with old generation 0 (kafk
a.coordinator.GroupCoordinator)
[2017-02-16 10:35:43,947] INFO [GroupCoordinator 0]: Stabilized group console-consumer-24139 generation 1 (kafka.coordinator.GroupCoor
dinator)
[2017-02-16 10:35:43,954] INFO [GroupCoordinator 0]: Assignment received from leader for group console-consumer-24139 for generation 1
 (kafka.coordinator.GroupCoordinator)
eventIdHd8c45b62-1550-4f28-9304-36f4a344e34ftimestamp1487259309006?{"id":0,"firstName":"tomcy","lastName":"john","dob":"1985-10-20"}
eventIdHf44c01dc-0ecb-45a1-8147-a85a766e89e1timestamp1487259309006?{"id":1,"firstName":"rahul","lastName":"dev","dob":"1989-08-15"}
eventIdHa061a57e-862f-44bb-b568-bad63b62f583timestamp1487259309006?{"id":2,"firstName":"pankaj","lastName":"misra","dob":"1982-08-10"}
eventIdH9f778d11-ce22-4335-b486-b4e67d95c574timestamp1487259309006~{"id":3,"firstName":"devi","lastName":"lal","dob":"1990-05-06"}
eventIdHd14be034-14e3-4c52-9380-5e5eed179757timestamp1487259309006~{"id":4,"firstName":"john","lastName":"doe","dob":"1992-06-25"}
```

图 6-27　通过 Kafka 日志查看 Consumer 运行细节

如图 6-27 所示，Interceptor 添加的头部信息已经作为消息的一部分，但是 Consumer 需要相应地处理消息的头部和主体信息。

案例四：内存 Channel、文件 Channel 和 Kafka Channel

到目前为止，在所有例子中都使用了内存 Channel。Flume 支持很多种 Channel，其中的文件 Channel 和 Kafka Channel 是最常用的。所有的 Channel 都可以完全相同的方式与 Source 和 Sink 连接，但是它们的定义有细微的差别。例如，如果定义一个文件 Channel，需要提供文件的位置和一些其他相关属性，并且还要确保文件位置具有足够的访问权限。

另一方面，我们定义一个 Kafka Channel，需要配置连接 Kafka URL 以及 topic 的信息，这是进行下一步消费的通道。因此，在这种情况下，拓扑结构可以是 Source → Kafka Channel → HDFS。

我们再次重复前面例子的变体，并观察它做了哪些改变：

（1）文件 Channel

1）复制 ${FLUME_HOME}/conf/spool-interceptor-kafka-flume-conf.properties 的内容到 ${FLUME_HOME}/conf/spoolfileChannel-kafka-flume-conf.properties 中，并把 Source、Channel 和 Sink 的配置改成下列配置：

```
agent.sources = spool-source
agent.sources.spool-source.type=spooldir
agent.sources.spool-source.spoolDir=/home/centos/flume-
data
agent.sources.spool-source.interceptors=ts uuid
```

```
# 时间戳 Interceptor 定义
agent.sources.spool-
source.interceptors.ts.type=timestamp

# UUID Interceptor 定义
agent.sources.spool-
source.interceptors.uuid.type=org.apache.flume.sink.sol
r.morphline.UUIDInterceptor$Builder
agent.sources.spool-
source.interceptors.uuid.headerName=eventId

# 以下是 Channel 定义
agent.channels = fileChannel
agent.channels.fileChannel.type = file
agent.channels.fileChannel.capacity = 100
agent.channels.fileChannel.transactionCapacity=10
agent.channels.fileChannel.dataDirs=/home/centos/flume-
data/flume-channel/data
agent.channels.fileChannel.checkpointDir=/home/centos/f
lume-data/flume-channel/checkpoint
agent.sources.spool-source.channels = fileChannel

# 每个 Sink 必须被定义
agent.sinks = kafkaSink
agent.sinks.kafkaSink.type=org.apache.flume.sink.kafka.
KafkaSink
agent.sinks.kafkaSink.brokerList=localhost:9092
agent.sinks.kafkaSink.topic=spooled-fileChannel
agent.sinks.kafkaSink.channel = fileChannel
agent.sinks.kafkaSink.useFlumeEventFormat=true
```

2）参照前面的例子将处理完成的文件重命名为 spool-1。

3）创建 Flume Channel 的数据文件，根据事务和回滚的需要，还需要创建检查点（checkpoint）目录：

```
mkdir  -p ~/flume-data/flume-channel/data
mkdir  -p ~/flume-data/flume-channel/checkpoint
```

4）确保该 Channel 的容量（配置文件中为 agent.channels.fileChannel.capacity=100）始终大于事务的容量（配置文件中为 agent.channels.fileChannel.transactionCapacity=10）。

5）使用以下命令再次运行 Flume 进程处理假脱机文件：

```
${FLUME_HOME}/bin/flume-ng agent --conf ${FLUME_HOME}/conf/
 -f ${FLUME_HOME}/conf/spool-fileChannel-kafka-flume-
conf.properties  -n agent -Dflume.root.logger=INFO,console
```

6）验证并观察：

- 使用相同的 Kafka 控制台 Consumer 命令来验证消息，所有消息必须与上例中的消息完全一样。

```
 ${KAFKA_HOME}/bin/kafka-console-consumer.sh
 ${KAFKA_HOME}/config/consumer.properties --topic
spooled-fileChannel -bootstrap-server localhost:9092 --
from-beginning
```

- 观察 Flume Channel 数据目录中的日志文件，并查看日志文件（~/flume-data/flume-channel/data/log-1）的内容，如图 6-28 所示。

```
?
/
eventId$53383d9d-1ded-4060-955a-8547ce1a40ae

1487264842472A{"id":0,"firstName":"tomcy","lastName":"john","dob":"1985-10-20"}/5
?
/
eventId$9f79c492-1d05-4008-8acc-7a3974abf28e

1487264842472@{"id":1,"firstName":"rahul","lastName":"dev","dob":"1989-08-15"}?]?
?
/
eventId$71bf0954-f3a9-4499-b86d-a2f2425b939f

1487264842472C{"id":2,"firstName":"pankaj","lastName":"misra","dob":"1982-08-10"}{4?
?
/
eventId$14b3233e-3652-442b-8230-b481d46ef165

1487264842472?{"id":3,"firstName":"devi","lastName":"lal","dob":"1990-05-06"m?
?
/
eventId$d10611c3-0d00-42c4-b58c-c71b1ab6cbc5

1487264842472?{"id":4,"firstName":"john","lastName":"doe","dob":"1992-06-25"}??
0
/
eventId$ecc94e4f-68c1-46dd-b400-30ebc0c14b74

1487264842472tamp
```

图 6-28　Flume 日志

（2）Kafka Channel

1）复制 ${FLUME_HOME}/conf/spool-interceptor-kafka-flume-conf.properties 的内容到 ${FLUME_HOME}/conf/spool-kafkaChannel-kafka-flume-conf.properties 中，并把 Source、Channel 和 Sink 的配置改成下列配置：

```
agent.sources = spool-source
agent.sources.spool-source.type=spooldir
agent.sources.spool-source.spoolDir=/home/centos/flume-data
agent.sources.spool-source.interceptors=ts uuid

# 时间戳 Interceptor 定义
agent.sources.spool-source.interceptors.ts.type=timestamp

# UUID Interceptor 定义
agent.sources.spool-
source.interceptors.uuid.type=org.apache.flume.sink.solr.mo
rphline.UUIDInterceptor$Builder
agent.sources.spool-
source.interceptors.uuid.headerName=eventId

# 以下是 Channel 定义
agent.channels = kafkaChannel
agent.channels.kafkaChannel.type
```

```
=org.apache.flume.channel.kafka.KafkaChannel
agent.channels.kafkaChannel.kafka.bootstrap.servers=localho
st:9092
agent.channels.kafkaChannel.kafka.topic=datalakeChannel
agent.sources.spool-source.channels = kafkaChannel

# 每个 Sink 必须被定义
agent.sinks = kafkaSink
agent.sinks.kafkaSink.type=org.apache.flume.sink.kafka.Kafk
aSink
agent.sinks.kafkaSink.brokerList=localhost:9092
agent.sinks.kafkaSink.topic=spooled
agent.sinks.kafkaSink.channel = kafkaChannel
agent.sinks.kafkaSink.useFlumeEventFormat=true
```

2）将处理完的文件重命名为 spool-1.log，并清空前面例子中使用的 Kafka 的日志文件。

3）使用以下命令再次运行 Flume 进程处理假脱机文件：

```
${FLUME_HOME}/bin/flume-ng agent --conf
${FLUME_HOME}/conf/  -f
${FLUME_HOME}/conf/spool-kafkaChannel-kafka-flume-
conf.properties  -n agent -Dflume.root.logger=INFO,console
```

4）验证并观察

- 观察配置的 Kafka topic 以及其队列深度（参见图 6-29）：

```
${KAFKA_HOME}/bin/kafka-run-class.sh
kafka.tools.GetOffsetShell --broker-list localhost:9092
--topic datalakeChannel
```

```
[centos@localhost kafka_2.11-0.10.1.1]$ bin/kafka-run-class.sh kafka.tools.GetOffsetShell
--broker-list localhost:9092 --topic datalakeChannel
datalakeChannel:0:6
```

图 6-29 日志记录的 Kafka topic 队列深度（Channel 相关）

- 使用相同的 Kafka 控制台 consumer 命令来验证消息，所有消息必须与上例中的消息完全一样。

```
${KAFKA_HOME}/bin/kafka-console-consumer.sh
${KAFKA_HOME}/config/consumer.properties --topic
datalakeChannel -bootstrap-server localhost:9092 --
from-beginning
```

正如观察到的那样，我们把 Kafka 用作一个 Channel，然后再次把这个事件写入 Kafka 的 topic 中。这乍看上去有误，但并非所有实际情况都如此，因为我们更倾向于将 Kafka Channel 中的事件放入处理流程或直接写入存储层。这个例子更多的是作为一个参考，指出一个非常重要的事实——Kafka 可以用作一个 Channel，而不是一个目标 Sink。这是一个巨大的优势，因为大多数近实时处理框架和批处理框架可以消费来自 Kafka 的事件。

在本章中，Flume 作为一个新的数据获取层组件，通过各种渠道获取丰富的客户信息。

此处已经涵盖了 RDBMS 和假脱机文件作为输入的情形，都可以将客户信息聚合到数据层中。数据获取层中已经有了 Sqoop 组件，这里又加入了 Flume 组件，整个数据获取层的示意图如图 6-30 所示。

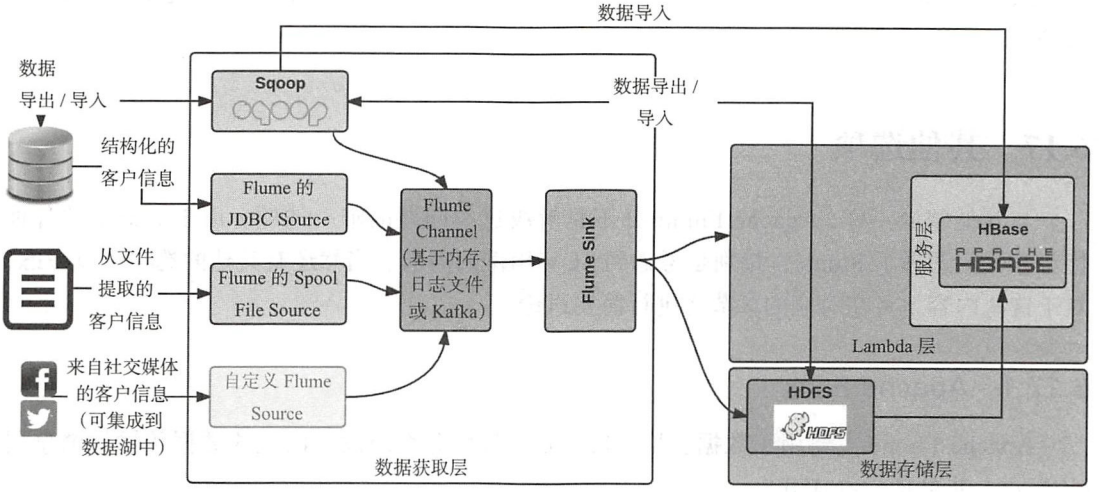


图 6-30　SCV 用例中的数据获取层与 Sqoop、Flume

6.15　适合使用 Flume 的场景

在选择 Flume 来处理不同的业务时，读者需要考虑的事项如下——遇到下列需求时可以选择 Flume：

- 从各种 Source 获取数据并存储到 Hadoop 系统中。
- 高速地处理大量数据到 Hadoop 系统中。
- 可靠地传输数据到目的地。
- 可扩展的解决方案，当数据涌入速度和数量增加时，只需增加机器就可以实现扩展。
- 架构中的各个组件可以动态配置，而无须启停服务。
- 整体架构的各种配置可以单点管理。

6.16　不适合使用 Flume 的场景

某些情形中，Flume 不是理想的选择。除了 Flume，还有其他的选项可以用来解决这些需求。下列场景中不推荐选择 Flume：

- 更偏重数据处理，而不是数据传输。这种场景更适合使用其他流式处理技术。
- 更偏重批量数据传输（常规批量而不是微批量）。

- 在不丢失数据的前提下需要更多可用的设置。
- 需要高可扩展性地对消息持久化（虽然没有一个科学的定量标准）。
- 有大量的消息消费者，这对 Flume 的可扩展性有很大的影响。

虽然 Flume 在很多情况下可以进行动态配置，但是某些配置更改（拓扑更改）还是会导致停机。

6.17 其他选项

与其他领域一样，Apache Flume 并不是解决这类问题的唯一选项。由于 Flume 的各种优势，我们选择了 Flume，特别是考虑到 SCV 用例的需求。当然还有其他的选项可以考虑，对于这些内容，将在对应的章节将进行简要讨论。

6.17.1 Apache Flink

Apache Flume 主要用于数据获取。Flume 将数据源系统发送的流式数据传输到消息层（用于进一步处理）和 HDFS。

为了将数据传输到 HDFS，Apache Flume 最适合用于流数据。然而，流式地获取数据，然后进行处理，也是 Apache Flink 的主要用途之一，并且它还拥有适用于这种场景的附加功能。

这并不意味着 Apache Flink 可以用于将数据传输到 HDFS，它确实有这种机制，但内置的功能存在不足之处。它有许多功能与 Flume 相同，但更多地用于流式处理方面。Flink 确实有一个滚动文件（rolling file）Sink，基于它可以将数据流写入 HDFS，但缺少很多与 Flume 内置功能类似的功能。

虽然 Apache Flink 日益流行，但其主要功能仍然是流式处理，Flume 仍将专注于数据获取领域，并将数据存储到 Hadoop 及相关的大数据存储系统。

6.17.2 Apache NiFi

按照 Apache NiFi 的官方网站描述的，Apache NiFi 是一个易于使用的、功能强大的、可靠的系统，用于处理和分发数据。

它是 Apache Flume 的代替品，并且具有功能丰富且易于使用的 Web 用户界面。它是一个高度可配置的、安全的、自底向上构建的、高度可定制（遵循可扩展性原则）的系统。

其中一个突出的特点是拥有可拖曳的 Web 界面，其中大多数可拖曳组件的属性可以动态配置。另一个显著的特征是它能够获取几乎所有的数据。如果用户有非常特别的数据，可以定制数据摄取方法。与输入数据类似，从 NiFi 向外发送数据也广泛地支持各种协议。同样地，输出方面也是可定制的，只要根据用户的需求编写一些自定义实现类即可。

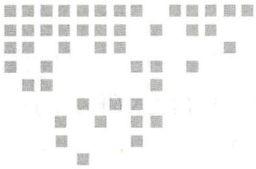
6.18　总结

首先，恭喜读者坚持到了这里。我们已经学习了数据湖的第一个功能模块 / 层（即数据获取层）中需要的技术。尽管只涵盖了两种技术（不能说已经深入研究了这些技术，但是用于我们的用例实现已经足够），在构建企业数据湖的进程中已经向前行进了相当远的距离。

本章与其他章节类似，首先着眼于这项技术在数据湖的整体架构中的位置，然后充分介绍了这项技术的细节，来说明为什么选择 Apache Flume 作为处理来自数据源系统的流式数据的技术。

接着，我们深入介绍了 Apache Flume，并开始研究 Flume 的主要概念和功能。我们也给出了一个 Flume 完整的工作示例，与 SCV 用例相一致。在结束前，我们着重分析了这项技术的优点和缺点，适用和不适用的场景。最后，我们介绍了一些其他的技术选项，如果确实合适，在实际建设数据湖时可以用它们代替 Flume。

阅读完本章后，读者应该对 Flume 有了一个整体的了解，并且准备好了一个完整的工作用例。现在，读者有了处理数据获取的方法，包括流式的和批量的，这些方法可以在企业中使用。



Chapter 7 第 7 章

使用 Apache Kafka 构建消息层

处理流式数据是数据湖中非常重要的一个课题。在本书讨论的数据湖架构中，处理流式数据是消息层的责任。在本章中，将详细介绍消息层，同时还会讨论该层实际的技术选型。

我们选择 Apache Kafka 作为应用于消息层的技术框架。本章深入探讨该技术及其在数据湖中的架构。

7.1 数据湖背景中的消息层

本章将讨论构成数据湖核心层（消息层）的一项技术。一个功能全面的消息层对于企业来说是至关重要的，它需要能够处理企业中来自不同应用程序的实时数据流。

本书选择了 Apache Kafka 来承担这一非常重要的任务，即流式数据处理。本章将介绍消息层的功能，并深入介绍 Kafka。

7.1.1 消息层

在第 2 章中，读者已经对消息层及其工作原理有了一个大概的认识，特别是在数据湖的场景中。图 7-1 再次展示了数据湖中消息层所处位置。

数据湖的消息层具有以下功能：

- 该层的一项核心功能是对源头（生产者，即 producer）和终端（消费者，即 consumer）进行解耦。
- 能够处理来自应用服务器节点每秒几百 MB 的高速信息流。
- 能够处理 TB 甚至 PB 级的数据。

- 能够低延迟、高吞吐地处理消息。
- 能够保证消息按照时间有序传输（持久性）向多个消费者传递相同消息的能力。在数据湖的场景中，同时向 Lambda 架构中的快速处理层和数据存储层提供相同的消息并实现不同功能。

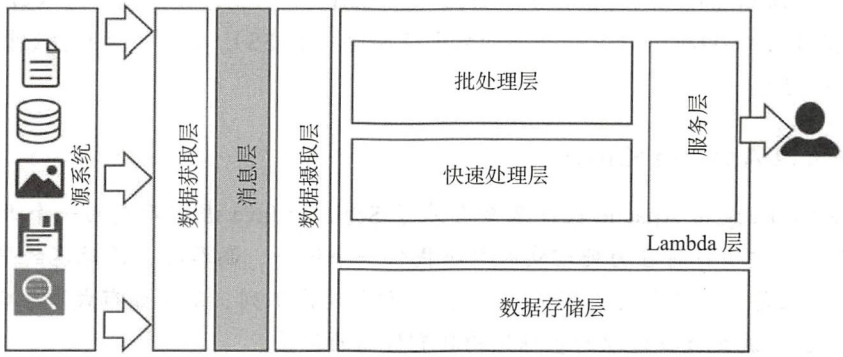


图 7-1　数据湖：消息层

- 能够对运营统计数据进行分析，聚合来自不同数据源的数据并分析。
- 借助廉价硬件实现高性能（诚然，这确实是一个需求）。
- 能够实现最低程度的数据增加与转换。

7.1.2　消息层——技术路线图

如果将数据比作高科技的血液，那么 Apache Kafka 就是血液循环系统。

——LinkedIn，Todd Palino

为了说明我们的用例并构建数据湖，这里采用 Apache Kafka 来承担消息层的职能。

图 7-2 呈现了贯穿全书的概念架构，并说明了与该概念架构对应的技术组件。本书将继续介绍每种技术以及它们与整体架构的关系，并在第三部分将所有技术融合在一起。

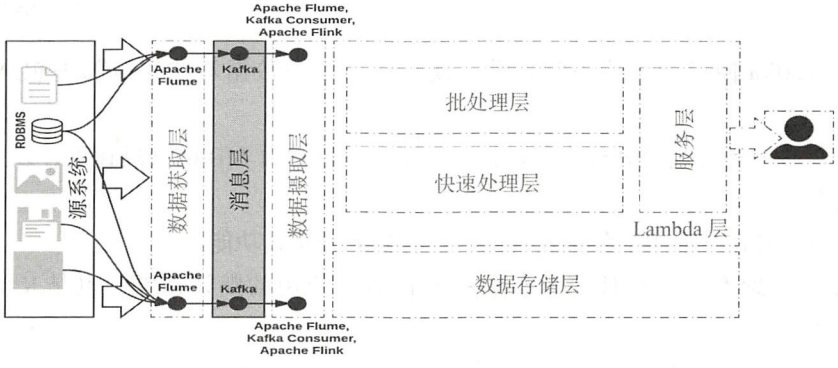


图 7-2　消息层技术路线图

就如同 SCV 用例，来自多个商业应用的实时数据将汇入 Flume，然后从 Flume，经由消息层，将数据写入 Hadoop 文件系统存储起来，同时进入 Lambda 架构中的快速处理层。我们将处理的数据是来自商业应用的实时用户行为数据，由用户与企业网站的交互行为产生。举例来说，诸如网页访问、链接点击、位置信息、浏览器信息等数据，都将会汇入 Flume。基于 Kafka 的发布（publish）、订阅（subscribe）功能，数据流将进入数据摄取层。数据摄取层将进行多目标分发，一条通向数据存储层（HDFS），而另一条则在数据摄取层本身进行必要的处理。

7.1.3 什么是 Apache Kafka

Apache Kafka 是由 Apache 软件基金会基于 Scala 和 Java 语言开发的一个开源的流式数据处理平台。该项目旨在为处理实时数据提供一个统一、高吞吐、低延迟的平台。其数据存储层基本上是一个“大规模可扩展的发布 / 订阅消息队列，基于分布式事务日志架构实现”，对于企业的流式数据处理的基础架构具有极高的价值。

——维基百科

本章下一节将介绍 Kafka 的更多技术细节。一言以蔽之，就像前面详述的那样，Kafka 提供了一个中间层解耦了数据源和数据消费者，同时提供消息层应有的数据处理能力。

7.2 为什么使用 Apache Kafka

我们采用 Apache Kafka 作为流式数据平台（也叫作 MOM（Message-Oriented Middleware，消息中间件）），关键原因是它的高可靠和低延迟的数据处理能力。

消息中间件（MOM）是支持消息在分布式系统之间进行发送和接收的软件或硬件基础架构。

——维基百科

Apache Kafka 拥有的一些关键特性，使其成为实现数据湖的功能的理想选择。具体如下：

- 可扩展性：能够高速处理大量数据。能实现每秒数百 MB 的处理速度，以及 TB 级的吞吐量。
- 分布式：Kafka 采用了分布式设计，具有如下分布式功能。
  - 副本：副本功能是任何分布式技术都需要使用的默认功能，也是 Kafka 的内置特性。
  - 分区能力：同理，分区能力也是分布式架构天然需要的特性之一。

- 高速：对这个特性的评价是相对主观的。一般来说，在市面上，Kafka 被认为是一款高性能的组件。
- 能够支持各种消费者组件。然而，由于消费者组件经常配置，需要执行大量处理工作，用户认为它们要慢一些。
- 天然的缓冲能力：为了适配各种消费者组件，Kafka 内置有缓冲能力。
- 发布 / 订阅特性：异步的发布 / 订阅集成模式。这是消息传输技术的关键特征之一。
- 接近数据库的可靠性：所有发送给 Kafka 的数据都是有序的，默认条件下是持久存储的。
- 容错性：基于分布式特性和副本，Kafka 内置有容错功能。

7.2.1　Kafka 简史

Kafka 最初是由 LinkedIn 的一个工程团队开发的，随后在 2011 年初，作为 Apache 软件基金会的一个开源项目发布。Kafka 是 LinkedIn 的消息传输平台。该项目于 2012 年 10 月 23 日升级成为 Apache 的顶级项目。

Jay Kreps、Neha Narkhede 和 Jun Rao 彼时就职于 LinkedIn 的工程团队，是 Kafka 的发明者。Jay Kreps 一直是著名作家弗兰兹·卡夫卡（Franz Kafka）的忠实粉丝，他认为 Kafka 这个名字适合这个消息传输平台，因为这个平台是以优化写操作为目的（引用自 Jay Kreps 在 Quora 的言论）。

Kafka 以新一代消息传输技术为首要的开发目标，如高吞吐量、低延迟以及高速处理大量实时数据的能力。Kafka 是用 Scala 语言开发的，也是 Apache 软件基金会（ASF）的顶级项目，有着强大的社区支持。Kafka 的设计大幅借鉴了 transaction log/commit log 机制，或者说受到了它的影响。

2014 年，Kafka 的创始人成立了自己的公司——Confluent，致力于开发以 Kafka 为核心的多种解决方案。该公司开发并维护多种 Kafka 的 connector，保持行业中的领先地位。

图 7-3 中概要显示了 Kafka 的发展历程。

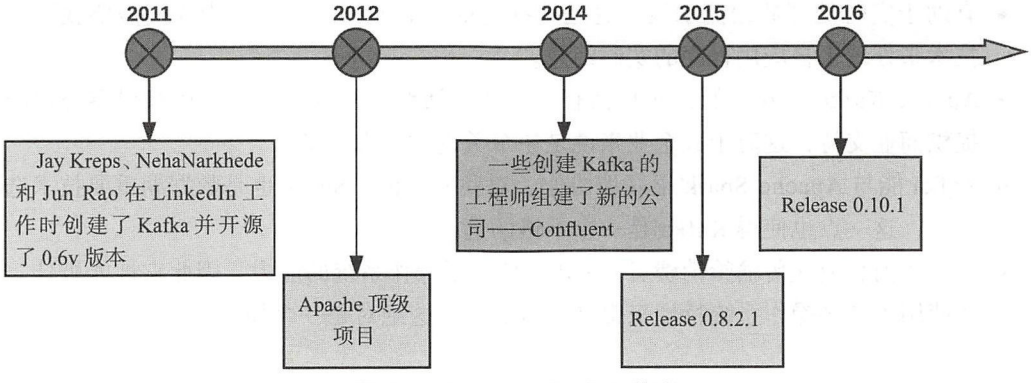


图 7-3　Apache Kafka 发展简史

7.2.2 Kafka 的优势

选择 Apache Kafka 是由于它在消息传递领域的优势。Kafka 拥有以下优点，使其成为构建数据湖的理想之选：

- 高吞吐：Kafka 可以高速处理大量数据，而对硬件条件没有特别要求。它能支持每秒数千条消息的吞吐量。
- 低延迟：按照大多数的用例需求，Kafka 能够以毫秒级的低延迟来处理这些消息。
- 容错性：Kafka 天然具有抵御集群内的节点 / 机器故障的能力。
- 持久性：数据 / 消息是存储在磁盘上的，使得它具有持久性，同时消息具有副本机制，永远不会丢失。
- 可伸缩性：通过添加额外的节点，Kafka 可以在不停机的情况下任意伸缩。Kafka 集群中消息处理是完全透明和无缝的。
- 分布式：Kafka 支持分布式架构，使其可以扩展使用副本和分区等功能。
- 消息分发（broker）能力。
- 高并发：每秒能够处理数千条消息，而且在低延迟条件下也能处理高吞吐量。Kafka 允许高并发地读取和写入消息。
- 默认持久性：默认情况下，消息是持久存储的，使得它具有持久性和可靠性。
- 对消费者友好：Kafka 可以与各种消费者组件集成起来。每个消费者处理消息的能力不同，由于 Kafka 的持久性，它可以与不同消费者集成使用。它也能与用各种语言开发的消费者组件很好地集成。
- 批量处理能力（类似 ETL 功能）：由于 Kafka 持久化消息，它也可以用于批处理类的用例，也可以做传统的 ETL 工作。
- 能够处理数据湖通常需要的各种用例，如日志聚合、Web 活动跟踪等。
- 专为用于商用硬件设计。对于 POC 来说很好，但是对于企业而言，由于多种原因，不推荐选择商业硬件，尽管它应该也能工作得很好。
- 有助于对数据处理流水线解耦。
- 有助于实时处理数据流水线。这是选择它的核心原因之一，因为我们需要找到一项技术来处理来自应用程序的实时消息。
- Apache Kafka 是开源的，而且拥有一个非常活跃的社区。另外，还有很多公司准备提供商业支持，这对于大企业来说是一个关键的考虑因素。
- Kafka 能与 Apache Spark 很好地协作。对我们来说，Spark 也是数据湖重要技术选项之一，这一点也使得 Kafka 是一个不错的选择。
- 为了将消息流式传输给消费者，Kafka 使用了操作系统的能力，因此它性能极佳。它还使用了许多操作系统特性来提高运行效率，这绝对是一个加分点。

非卖品，仅供非商业用途或交流学习使用

7.2.3　Kafka 的劣势

虽然 Kafka 的优势远胜过它的不足，但是最好对它的局限性有所了解，并且只有当它优势大到可以完全掩盖其劣势时才考虑使用它。下面列举的是 Kafka 的主要缺点，其中一部分可能只与特定用例相关，但与我们的实际用例无关。

- 没有一整套的管理和监控工具。从长远来看，这使得企业的技术支持人员对于选择 Kafka 存有一丝担心。
- broker 会使用某些系统调用向 consumer 传递消息，但如果需要对消息进行一些处理，就会显著降低 Kafka 的性能。反之，如果消息没有变更，性能就没问题，因为它使用系统既有的功能。
- Kafka 仅匹配确切的 topic 名称，不支持用通配符匹配 topic，这使得它无法支持某些场景。
- 其他语言所需的 API 由不同的个人和企业维护，这可能会导致问题出现，因为这些维护者更新 connector 的速度可能较慢。
- Kafka 的 broker 及其方法往往被认为过于简单。因此，需要用其他组件来满足某些要求，如 ZooKeeper（状态协调）和 MirrorMaker（群集间通信），这使得对整体架构的部署和支持成为一场噩梦，尤其对于技术支持人员而言。
- 本质上，Kafka 根本不关心单条消息的大小。但是，随着消息的体积增加，broker 和 consumer 开始压缩这些消息，因此，当数据在管道中流动，解压缩和压缩发生时，节点中的内存使用效率将会降低。这可能会影响吞吐量和性能。
- 用户经常批评，随着 Kafka 集群中队列数量的增加，它的性能和反应速度开始下降。Kafka 将这个问题归咎于设计。
- Kafka 缺乏对其他消息系统范式的支持，如请求 / 响应（request/reply）、点对点队列（point-to-point queue）等，这使得它在某些用例中存在问题。

7.3　Kafka 的架构

本节重点介绍 Apache Kafka 的架构。首先将深入介绍 Kafka 的核心准则，随后会详细介绍其架构的各个组件。

所以，让我们一起前进吧。

7.3.1　Kafka 架构的核心原则

LinkedIn 的工程团队开发了 Kafka，其背后的主要动因是：

为了创建一个统一的消息传输平台，以适应来自大型组织中各个应用程序的实时数据传输。

——LinkedIn

非卖品，仅供非商业用途或交流学习使用

Kafka 在构思和设计上有着其核心的架构准则。这些准则概括如下：

- 追求性能极致（例如使用了压缩算法和 B-tree）。
- 尽可能简化不必要的功能来实现性能优化（例如使用零拷贝（zero-copy）和直接使用 Linux 文件系统缓存）。
- 分布式架构。
- 容错性。
- 消息持久化存储。
- 尽可能剥离冗余的功能。
- 将部分功能交给消费者应用管理，比如由消费者组件管理消息的状态，也不提供 ACK 机制。
- 可扩展性：允许应用程序以多种方式接入和使用 Kafka。另外，如果需要，Kafka 提供了开发新的 connector 的方法。
- 高吞吐量（处理大量消息的能力）。
- 实时处理这些消息并实时输出处理后的消息。
- 低延迟传输。

7.3.2 数据流的生命周期

有了 Apache Kafka 这项技术，让处理数据流成为可能。在介绍 Kafka 的功能之前，先看看数据流动的整个生命周期中的各个事件。Rassul Fazelat 在 LinkedIn 的博客中详细解释了下面的数据流图（参见图 7-4）。

如图 7-4 所示，数据流的生命周期由 3 个环节组成，每个部分都有一个明确的功能：

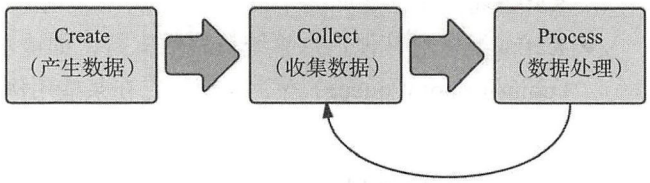


图 7-4 数据流的生命周期

- Create：最重要的环节。它从各种内部业务应用程序、外部合作伙伴以及其他应用程序产生数据流。例如，托管业务应用程序的服务器日志，以点击流的形式从各种业务应用程序收集的行为数据，页面浏览数据，来自社交网络的社交数据，各种传感器（IoT）发送的数据等。
- Collect：该环节帮助收集这些数据，并使得这些数据可以被处理。这部分功能就是由 Apache Kafka 实现的。当然也有其他选项可以实现这个功能，比如 ActiveMQ、HornetQ 等。
- Process：处理流式数据并产生有意义数据流以备分析之用。数据湖的架构中也需要这项功能，下一章将深入介绍一种技术，该技术能提供这项功能。这个领域的一些代表性技术是 Apache Spark、Apache Flink 等。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

7.3.3　Kafka 的工作原理

Kafka 的架构本质上非常简单，有一些非常重要的组件，它们是整个 Kafka 运作的关键。图 7-5 显示了 Kafka 的工作原理，所有的组件都被标识出来了。构成 Kafka 架构的主要组件如下：

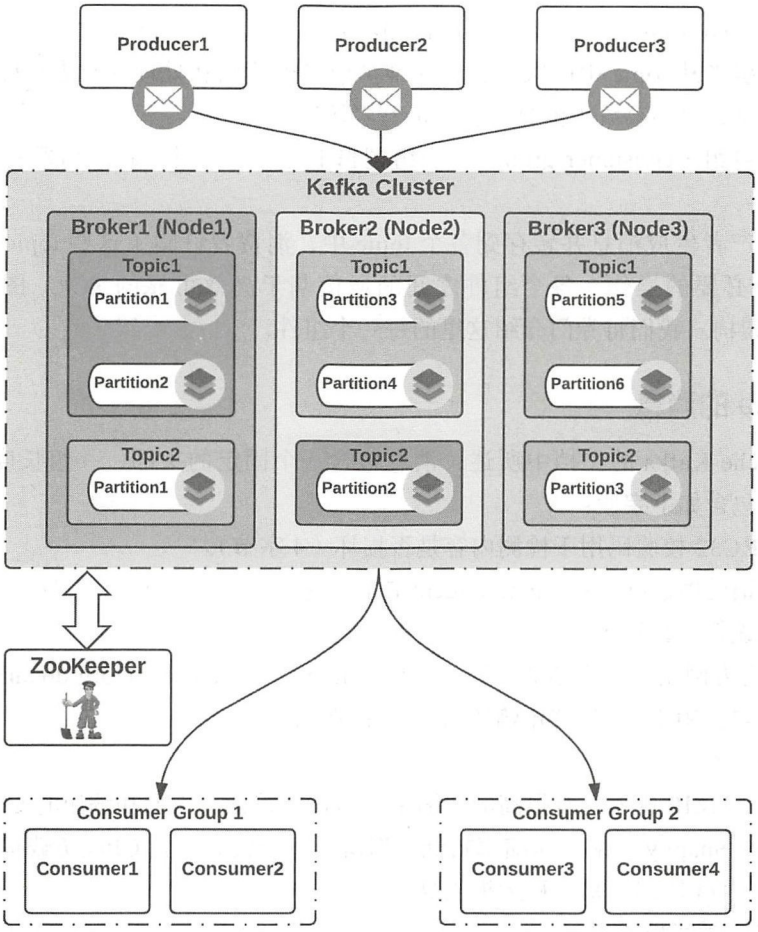


图 7-5　Kafka 架构

- 消息（message）：从生产者生成，通过位于 broker 中的 topic 传输到消费者那里。
- 生产者（producer）：顾名思义，就是生成这些流式数据（消息）并将其写入某个 topic 中。
- topic：生产者写入消息的类别。生产者产生的属于某个特别类别的数据会进入该 topic。理想情况下，Kafka 会为每一类消息创建一个新的 topic。
- broker：Kafka 集群包含多个服务器 / 节点。每个节点都是一个 Kafka broker，包含多个分区和多个 topic。
- 分区（partition）：每个 Kafka 中的 topic 都是分区的，每个分区都包含来自生产者

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

的有序消息。对每个分区的写入在底层都是连续的，这是 Kafka 高性能的主要原因之一。在写入消息到 topic 中的某个分区时，每个消息都被赋予一个称为偏移量（offset）的序列号，并用该序列号作为分区中消息的唯一标识。Kafka 天然就是分布式的架构，将这些 topic 的分区分布在多个 broker 中，以提供系统的容错性和消息的持久性。

- 消费者（consumer）：顾名思义，订阅特定类别（某个 topic）并消费该 topic 中的消息。消费者从 topic 中读取消息，并维护消息被读取和处理的状态。Kafka 既不删除这些消息，也不会维护消息被各个消费者读取的状态。
- 消费者群组（consumer group）：一组逻辑上分在一组的 Kafka 消费者被称为消费者群组。

所以，生产者生成消息并转存到各个 topic 中，消费者订阅了这些 topic，就能读取这些消息并进行必要的操作。各个组件之间的通信基于高效可靠的 TCP。图 7-5 所示就是 Kafka 的整体架构。我们将专门介绍这里的每一个组件。

7.3.4 Kafka 的消息

正如 Apache Kafka 的文档中所述，消息包含一个固定的头部、一组长度均可变的键 / 值对。头部中包含如下字段：

- 一个 CRC32 校验码用于检测内容是否损坏（4 字节）。
- 一个所谓的魔法标识符（magic identifier），它的值为 0 或 1（1 字节）。
- 属性标识符（1 字节）。

Kafka 的官方网站上有详细的文档，网址为 https://kafka.apache.org/documentation/。

  - Bit 0-2：用于压缩的编码解码器，可以取以下值。
    - ◆ 0：不压缩。
    - ◆ 1：GZIP 压缩，有很高的压缩率，但性能较低，压缩和解压时 CPU 负载很高。
    - ◆ 2：Snappy 压缩，压缩率较低，但解压性能较高，且 CPU 负载很小。
    - ◆ 3：lz4 压缩，是一种无损压缩算法。
  - Bit 3：时间戳类型。
    - ◆ 0：创建时间。
    - ◆ 1：追加日志的时间。
  - Bit 4-7：保留位。

- 一个可选的时间戳，当且仅当魔法标识符大于 0 时有效（8 字节）。

消息的正文部分包含以下字段：

- key 的长度（4 字节），假设长度为 K。
- 实际的 key（K 字节）。
- value 的长度（4 字节），假设长度为 V。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

■ 实际的 value（V 字节）。

文档中已经说明，key 和 value 的长度是开放的，因为已经对消息格式进行了大量的优化，而且这种设计使得 Kafka 的适应性非常好。

图 7-6 显示了 Kafka 消息的格式，涵盖了刚刚介绍的所有方面。

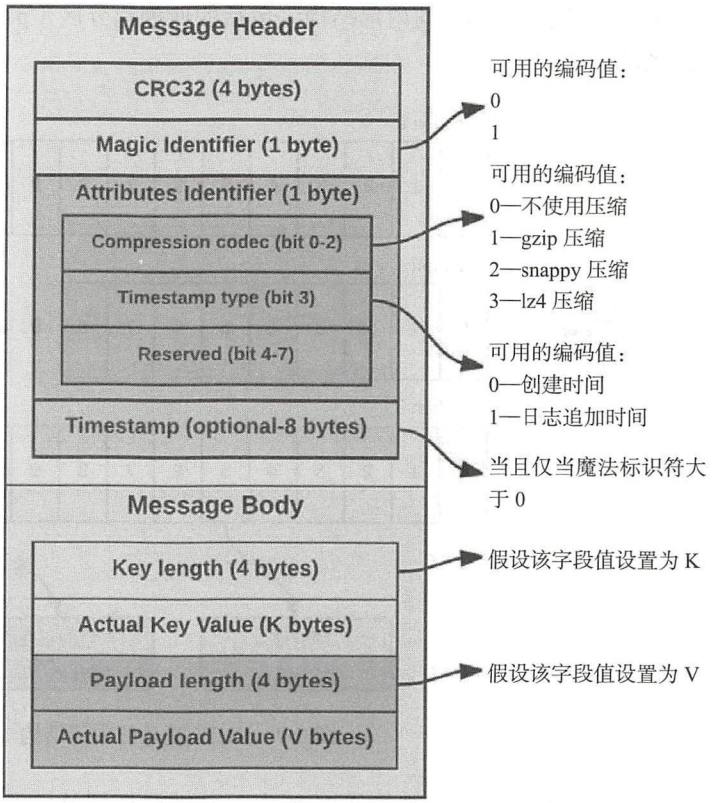


图 7-6　Kafka 消息格式（源自 Kafka 官方文档）

7.3.5　Kafka 生产者

顾名思义，生产者指的是产生消息（流式数据）到 Kafka 集群的客户端，这些数据会进入到指定的 topic。存在不同类型的生产者，它们有着不同的发送消息的方式。有些生产者产生消息的速度快，但消息的体积较小（消息的内容较少但吞吐量很高）。而另外有些生产者产生消息的速度较低，但消息的体积巨大（吞吐量较低但消息体很大）。

生产者可以选择以同步或异步的方式发送消息。Kafka 支持异步处理，允许生产者发送微批量的消息，以此减少生产者和集群之间的延迟。某些情形下，这会非常有用，具体取决于用例情况。如果需要实时处理消息，这种微批量处理方式将是最好的选择，因为它能减少消息生成和实际的处理 / 分析之间的延迟。

7.3.6　Kafka topic 中的数据持久化

Kafka 提供了高级的抽象，由生产者推送的数据会被持久化存储起来，同时有一个明确的名称。为了适应副本和容错的机制，每个 topic 被分为多个分区，每个分区分布在 Kafka 集群不同的 broker 中。

topic 中并没有记录消息者是否已经消费了某条消息，而是将监控的控制权完全交给了

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

消费者，由消费者来跟踪消息的消费状态。消费者从某个 topic，以某个特定的消息 ID 开始接收消息，这种机制必须由消费者来管理。

图 7-7 显示了 topic 的内部结构，包括如何创建分区（partition）以及消息是如何进入各个分区的。

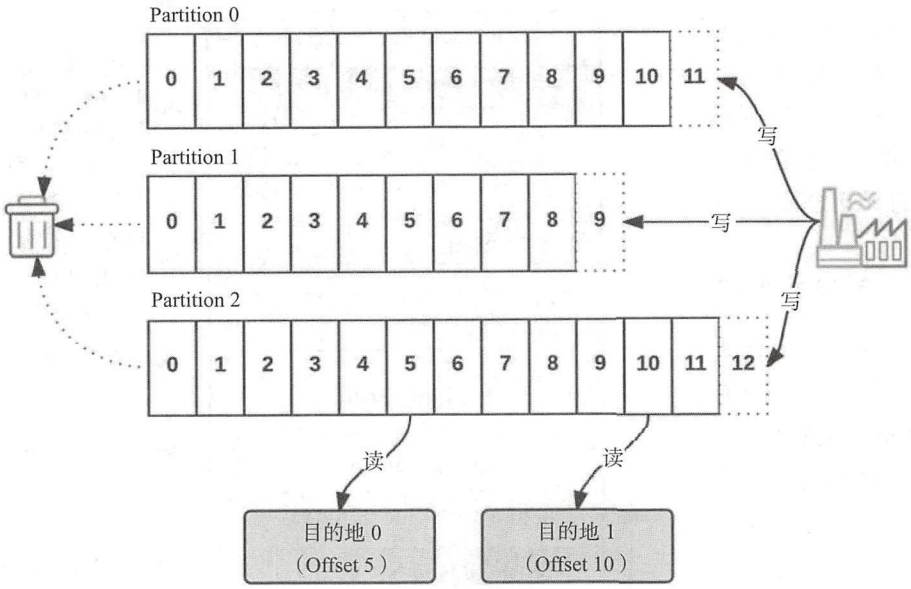


图 7-7 Kafka 消息的内部结构

如图 7-7 所示，消息被传递给 topic，并被追加到分区的尾部。topic 中的每个分区在逻辑上都是一个日志文件。

即使所有的消费者都已经消费这些消息，Kafka 仍然会将这些消息保留在分区中，然后会以不同的方法从分区中移除这些消息。Kafka 定义了一些属性，通过配置这些属性，用户可以选择如何移除这些消息。表 7-1 中罗列了这些属性。

表 7-1 Kafka 消息移除配置

描 述	配 置 属 性
partition 中消息的数量	log.flush.interval.messages
时间戳	log.default.flush.interval.ms，log.flush.interval.ms
partition 中所有消息的数据体积	log.retention.size

7.3.7 Kafka 中 topic 的划分：partition

为了实现容错性和副本机制，Kafka 的 topic 被划分为多个 partition，并分布在多个 broker 节点上。每个 partition 选择一台服务器（broker 节点）作为 Leader，另外有 0 或若干台服务器（称为 Follower）接受该 Leader 的领导。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

Leader 负责读写数据的操作。Follower 是 Leader 的副本，并总是与 Leader 保持数据同步。任何分布式系统都有类似的机制，如果 Leader 因任何原因失败，它的某个 Follower 会自动接管起来，并成为 Leader。Kafka 中的这种机制可以实现无缝的容错性和消息持久化。

为了将整个 Kafka 集群中的 Leader 和 Follower 分开，Kafka 将 partition 分布最优的节点选为 Leader，其他节点作为 Follower。同理，这个过程对于用户来说是无缝衔接的，它是由 Kafka 架构在内部处理的。

7.3.8　Kafka 中的消息 broker

broker 是 Kafka 中至关重要的组件，它实际上为 Kafka 提供了解耦能力。生产者产生的消息被传输到 broker 中的 topic 中，消费者会消费来自该 topic 中的消息（消费者会订阅 broker 中的 topic，broker 会向该消费者发布消息）。这些消息被发布到这个 broker，且可以由多个 broker 节点组成了 Kafka 集群。Kafka 的数据存储由 broker 负责。broker 同时负责从该 broker 内部的 partition 中查找消息。这种机制是必需的，因为消费者需要向订阅的 topic 请求消息。

消息存储在由多个 partition 组成的 topic 中，这些 partition 会被复制到集群的多个 broker 中。图 7-8 所示是一个复制的示例图，这个 Kafka 集群有 4 个 broker（图 7-8 只是一个示意图，内部实现参考 Kafka 的具体实现）。

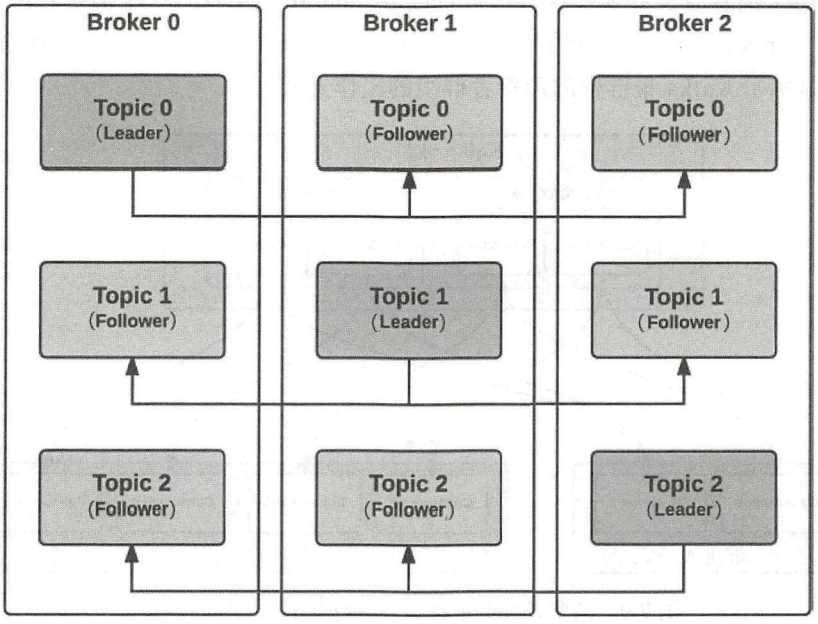


图 7-8　消息副本在 Kafka 集群 broker 之间散布

图 7-8 中，每个 partition 被复制到多个节点中。为了清晰起见，就不显示 topic 中的

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

partition 信息了。

Kafka 集群中的某个 broker 承担了集群控制者的角色，并管理这些 partition 的 Leader 和 partition 的状态。另外，视情况而定，该 broker 还负有重新分配 partition 的职责，并根据配置安排副本。

7.3.9 Kafka 的消费者

消费者会消费 Kafka 集群中某个 topic 的消息。消费者按照有序的方式消费该 topic 的各个分区中的消息。消费者需要确保它能跟踪从该 topic 获取 / 读取的消息，并且在拉取消息时需要提供顺序 ID，并随后以有序方式读取该 ID 对应的消息。消费者使用拉取数据的机制来消费消息，这是 Kafka 能够实现高吞吐量的消息消费的原因。此外，消息消费的吞吐量取决于各个独立的消费者，因此 Kafka 可以兼容性能各异的消费者。

根据需要，消费者可以随时回退消费旧消息，因为按照 Kafka 配置的属性，消息是存储在集群的内存中的。

在一个消费者群组中，需要保证消息至少被消费一次。

消费者群组

消费者群组允许多个主机组成一个群组（使用同一个群组名称）来访问给定的 topic。消费者群组机制保证消息只被该群组中某一个消费者消费到。如图 7-9 所示，分区将分配给群组中的每个消费者，并委托这些消费者读取 topic 中的消息。一个消费者可能会分配到一个或者多个分区，但需要注意的是，一个分区只能分配给该群组中的某个消费者。

图 7-9 显示了 Kafka 集群中的消费者群组的工作方式。

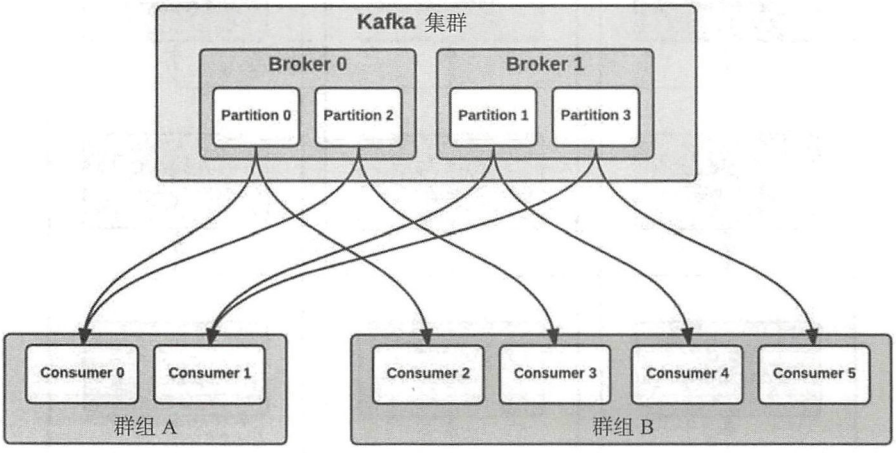


图 7-9 消费者群组是将多个消费者逻辑上划归一组

如图 7-9 所示，为了便于读者理解消费者群组的概念，这里只采用了 2 个 broker，1 个 topic，每个 broker 中包括 2 个分区。同时给定了 2 个消费者群组，群组 A 包括 2 个消费

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

者，群组 B 包括 4 个消费者。群组 A 的消费者们从 broker 0 和 broker 1 中的各个分区中读取消息。群组 B 的部分消费者从 broker 0 中的分区中读取消息，另一部分消费者从 broker 1 中的分区中读取。

7.4　其他 Kafka 组件

此外，Kafka 集群中还有一些重要的组件，这些组件是 Kafka 正常运行所必需的。下面列举了若干重要的组件：

- ZooKeeper
- MirrorMaker

7.4.1　ZooKeeper

ZooKeeper 是一个非常重要的隐形组件，它是 Kafka 正常运行所（强制性的）需要的。ZooKeeper 将起到以下作用：

- 负责将每个 broker 引入集群，成为集群成员。
- 在集群内选举 Kafka 控制者，执行一些重要的功能，比如管理分区及其副本的状态。
- 完成 topic 的配置，如分区的数量，选择主分区、分区副本位置等。
- 维护每个 broker 中的访问控制列表和配额。

7.4.2　MirrorMaker

顾名思义，MirrorMaker 用于维护跨集群的镜像数据。如图 7-10 所示，该组件可用于将整个集群从一个数据中心镜像到另一个数据中心。

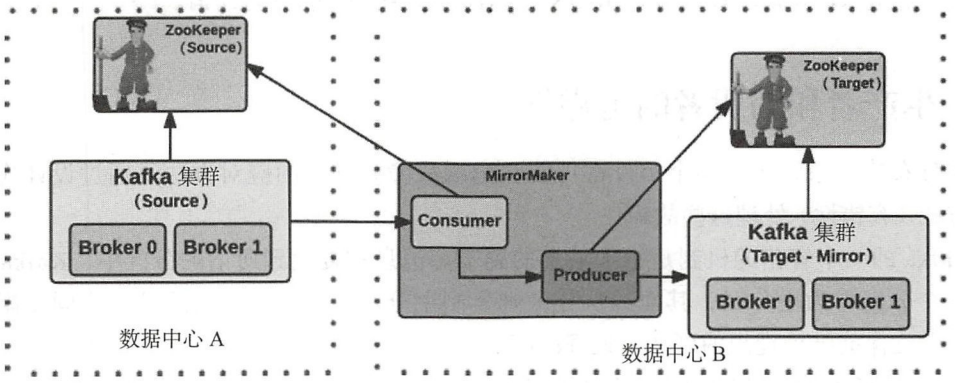


图 7-10　MirrorMaker 对 Kafka 集群执行跨数据中心的镜像操作

本质上，该组件同时使用消费者 API 和生产者 API 来镜像集群。它从一个或多个源集群读取消息，然后写入目标集群。

仅供非商业用途或交流学习使用

7.5 Kafka 编程接口

Kafka 包含两类编程接口机制：

- 底层核心 API。
- RESTAPI——为了方便使用，对底层核心 API 进行 REST 封装。

7.5.1 Kafka 核心 API

这些核心 API 在 Apache Kafka 的文档中有详细的记录：

- 生产者（producer）API：包括一组 API，使得用户可以向 Kafka 集群中的某一个或者多个 topic 发布流式数据。
- 数据流（stream）API：包括一组流式数据处理相关 API。基于这些 API，用户可以根据需求处理流式数据，将其从现有格式转换成指定格式。对比现有的生产者和消费者 API，数据流 API 是较新的 API。
- 连接（connect）API：这些 API 使得 Kafka 可扩展，其中的方法可以用来构建 Kafka 连接器，用于向 Kafka 输入或者输出数据。
- 消费者（consumer）API：包括从 topic 中订阅数据相关的 API。由于消费者处理消息并记录消费情况，基于这些 API 可以精确地从分区中消费消息。

7.5.2 Kafka REST 接口

REST 接口封装了核心 API，由 Confluent（创建 Kafka 的前 LinkedIn 工程团队成立的公司）创建并维护。使用这些 API 的确会产生性能损益，因为这种机制完成一次功能需要两级调用。同时，还需要一个额外的服务器来托管这些 REST 的 API。另外，每次调用这些 API，解析 JSON 请求以及创建 JSON 响应都会带来额外的时间开销。

7.6 生产者和消费者的可靠性

在分布式的系统中，各个组件都可能会出现故障。常见的应对方法是通过设计代码以无缝方式（容错性）处理这些故障。

Kafka 的一个容错的机制是维护消息的副本。消息被复制到所谓的分区中，Kafka 自动选择一个分区作为主分区，其他的从分区则复制主分区。主分区还会维护一个同步副本的列表，以确保能维护理想的副本来处理故障。

生产者将消息发送到 topic（位于 Kafka 集群中的 broker）中，并且配置生产者的 request.required.acks 属性来管理持久性，该配置具有以下取值：

- 0：消息写入网络 / 缓冲区。
- 1：消息直接写入主分区。

- all：当所有同步副本（ISR）都获得消息时，生产者将收到一个回执。

消费者从 topic 中读取数据，在 Kafka 中，消息读取的状态由消费者本身来保持，而不是 Kafka。这种机制使得 Kafka 不需要管理消费者对消息消费的情况。每个消费者需要自行管理消费者的偏移量（消费者最后一次读取的消息的序列 ID）。topic 中的消息保持不变，在消费者订阅 topic 以及读取该消息后也不会立即删除。Kafka 会根据 broker/topic 的配置情况，从 topic 中删除消息。因此，即使消费者已经宕机或无法正常消费消息，消息仍然保留在 topic 中，假如保留期限设置在合理的水平，当消费者上线时，使用它的偏移量，可以从该偏移量的开始读取消息，而不会造成其他影响。这是 Kafka 实现消费者可靠性的机制。

图 7-11 显示了消费者在遍历 broker 中某个 topic 分区时的各个位置。

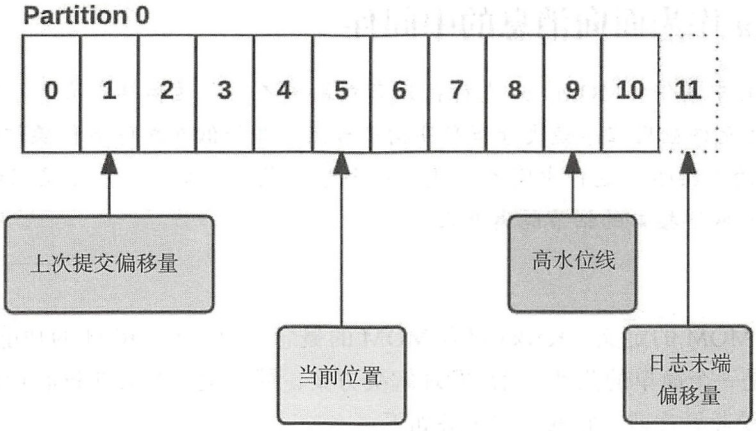


图 7-11　消费者使用偏移量追踪消息的消费情况

关于消费者维护的一些重要位置的总结如下：

- 上次提交偏移量（last committed offset）：上一条写入日志的消息的偏移量。如果分区出现故障且消费者被挂载到新的分区，则以此偏移量作为消费起点。
- 当前位置（current position）：消费者当前读取消息的偏移量。
- 高水位线（high watermark）：已成功复制到群集内所有副本的消息的偏移量。该消息需要在消费者正常读取其他消息之后才能被消费，然后水位线偏移量向前移动。
- 日志末端偏移量（log end offset）：生产者最后写入的消息的偏移量。

7.7　Kafka 的安全性

在 LinkedIn 设计和开发 Kafka 时，在很大程度上忽视了安全性。当 Kafka 成为 Apache 的主要项目之后，才开始考虑安全性的问题。在 2014 年下半年，Kafka 的各种安全性讨论才逐渐提上日程，尤其是静态数据安全和传输层安全。

Kafka 的 broker 允许客户端连接多个端口，并且每个端口支持不同的安全性机制，具

体如下：

- 无加密和认证。
- SSL：加密并认证。
- SASL：Kerberos 认证机制。
- SSL+SASL：用 SSL 做加密，用 SASL 做认证。
- 采用类似 UNIX 权限机制来认证客户端的读 / 写权限。

这些安全是的特性是由 Confluent 主导开发的，详细的文档可参见 http://docs.confluent.io/2.0.0/kafka/security.html。

7.8 Kafka 作为面向消息的中间件

面向消息的中间件（MOM）是支持在分布式系统之间发送和接收消息的软件或硬件基础架构。MOM 允许应用程序模块分布在异构平台上，并降低开发跨多种操作系统和网络协议的应用程序的复杂性。这种中间件构建了一个分布式通信层，它能把应用程序开发者与多种操作系统和网络接口的细节隔离开来。

——维基百科

基于上述 MOM 的定义，Kafka 符合 MOM 的概念，也满足 MOM 的功能需求。但是，Kafka 不仅仅是一个简单的队列 / 消息管理解决方案，同时也具有某些核心功能，使其比传统的 MOM 更受业界欢迎。它的一些优势如下：

- 基于零拷贝的数据日志方法（分布式提交日志），且消息总是追加到日志。
- 大量使用分区来在集群中实现跨 broker、分区、topic 的消息分发。
- 原生分布式架构，有助于容错和消息持久化。
- 内置副本机制。
- 原生可伸缩性。每个分区分配一个消费者，为了伸缩可以持续增加分区。
- 因为消息的状态由消费者而不是 broker/topic 管理，所以即使订阅消息的消费者下线或者不可用，这个队列机制依然可以正常工作。
- 持久化消息：由于 Kafka 坚信将数据持久化到磁盘中不存在性能问题，因而保证了消息的持久性，这是很多传统的消息系统所不具备的特性。同时，消费者可以按照自身的节奏来消费这些消息，而不依赖生产者的速度。

就目前而言，Kafka 不仅能胜任 MOM 的角色，还能提供诸多核心功能，这些功能正是大数据平台和现代而非传统消息系统所依赖的功能。

随着社交网络产生的数据越来越多以及物联网的迅速发展，会出现越来越多的流式数据。Kafka 的各项天然的特性，使得它能够对这些流式数据做出有意义的处理，用以解决企业中的商业问题。

零拷贝描述的是一种计算机操作，在这种操作中，CPU 不会执行将数据从一个存储区复制到另一个存储区的任务。当通过网络传输文件时，这样通常会节约 CPU 时钟周期和内存带宽。

——维基百科

7.9　Kafka 与水平可扩展架构

本章在前文中涵盖了 Kafka 的主要工作准则，这里不再赘述。下面列举了 Kafka 能实现可水平扩展架构的一些关键因素：

- 分区：分区的可伸缩机制是指 topic 分割成多个分区，分区可以增加。
- 分布式：集群中可以有一个或者多个 broker，可以通过增加 broker 来实现扩容。
- 副本机制：与分区类似，消息的多个副本用于容错，同时也为 Kafka 增加了可伸缩性。
- 可伸缩性：每个消费者从单个 topic 的单个分区读取消息，消费者可水平扩展，并且让新添加的消费者从新分区读取消息（一个消费者只能从同一分区读取消息，这是一条规则），如图 7-12 所示。

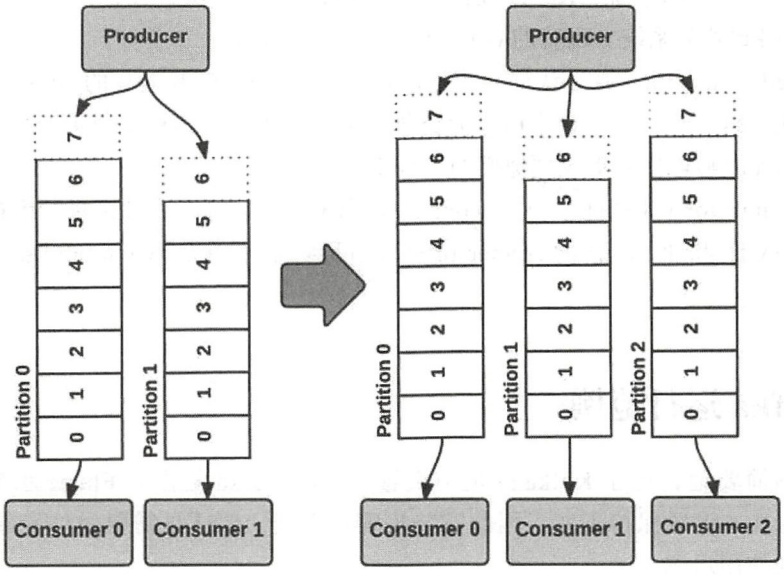


图 7-12　通过添加消费者实现水平扩展

7.10　Kafka 连接

可扩展性是 Kafka 严格遵循的重要设计原则之一。Kafka 的连接工具使得 Kafka 具有

了可扩展性。该工具使 Kafka 能够与外部系统连接，将数据输入其中，并将数据输出到其他系统。它有一个通用的框架，可以基于它编写自定义 connector。关于 Kafka 连接的更多细节可以在 Kafka 官方文档中查阅，参考网址如下：https://kafka.apache.org/documentation.html#connect。

图 7-13 显示了 Kafka Connect 的工作原理。

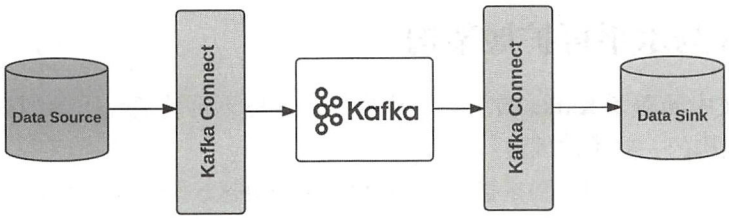


图 7-13 Kafka Connect 工作原理

Kafka 的 connector 分为两类：

- Source connector：输入数据到 Kafka 的 connector。
- Sink connector：将 Kafka 中的数据输出到外部系统的 connector。

为了适应各种外部系统，Kafka 提供了大量 connector 可供选择，使用这些 connector，Kafka 可以连接到外部系统。现有的 connector 又分为两大类：

- certified connector（通过认证）：使用 Kafka Connect 框架编写的 connector，并且符合最佳编码实践。它们通常由专业提供商创建和维护。这些提供商开发 connector 使得 Kafka 能够连接并集成到他们的工具中。
- 其他 connector（非认证）：这些 connector 尚未通过认证，由社区进行维护和管理。

完整的认证和非认证的 connector 列表可参见：https://www.confluent.io/product/connectors/。

7.11 Kafka 运行范例

本书此前简要地讨论了 Kafka 的基本设置，当时 Kafka 是作为 Flume 示例的一部分。由于 Kafka 的基本设置保持不变，因此安装步骤将不变，但还是需要另外考虑 Kafka 作为消息 broker 的使用示例。

目前 Kafka 最适合的编程语言是 Scala 或者 Java。因此，为简单起见，在示例中我们选择使用 Java 语言。

7.11.1 安装

1）使用以下命令下载 Kafka 的二进制文件：

```
wget http://redrockdigimark.com/apachemirror/kafka/0.10.1.1/kafka_2
.11-0.10.1.1.tgz
```

2）切换到想要安装 Kafka 的用户目录，使用以下命令提取 tar 包的内容，并将提取 Kafka 的目录记为 ${KAFKA_HOME}：

```
tar -xzvf ${DOWNLOAD_DIRECTORY}/kafka_2.11-0.10.1.1.tgz
```

3）使用以下命令将 KAFKA_HOME 设置为环境变量，并将其添加到 ~/.bashrc 中：

```
export KAFKA_HOME=<PATH to KAFKA Directory>
export PATH=$PATH:$KAFKA_HOME/bin
```

4）切换到 Kafka 所在文件夹，即 ${KAFKA_HOME}，然后运行以下命令启动 Kafka 服务：

```
${KAFKA_HOME}/bin/kafka-server-start.sh
${KAFKA_HOME}/config/server.properties
```

5）该 Kafka 实例基于 Kafka 二进制文件中的默认配置文件 server.properties。

Kafka 安装包中包括了生产者、broker（服务器）以及消费者的默认配置。我们将通过工作示例进行更深入地了解。

7.11.2　生产者：向 Kafka 写入数据

Kafka 的默认生产者配置位于 ${KAFKA_HOME}/config/producer.properties，内容如下：

```
############################## Producer Basics ##############################

# broker 列表，用于在初始化时发现集群其余节点
# 格式: host1:port1,host2:port2 ...
bootstrap.servers=localhost:9092

# 确定应用在生成数据之上的数据压缩编解码器: none, gzip, snappy, lz4
compression.type=none

# 用于对事件分区的 partitioner 类名；默认情况下随机将事件分到各个分区
# partitioner.class=

# 客户端等待请求响应的最长超时时间
# request.timeout.ms=

# KafkaProducer.send及KafkaProducer.partitionsFor 阻塞时长
# max.block.ms=

# producer 将会等待给定的延迟时间以允许其他消息记录发送，这些消息记录可以批量处理
#linger.ms=

# 请求允许的最大字节数
#max.request.size=

# 批量记录发送到一个分区时，默认的批处理 size，以字节为单位
```

```
#batch.size=

# 生产者可以用来缓冲发送至服务器的记录的内存字节数
```

以上是 Kafka 生产者所需的基本配置，其中最为重要的配置参数如下：

1）bootstrap.servers：指的是 Kafka broker 监听的端口，以逗号分隔来指定 Kafka 集群中的多个 broker。

2）key.serializer：指定用于序列化消息中 key 的程序。它的取值是序列化程序的完整类名。例如，控制台生产者的默认取值如下。

```
org.apache.kafka.common.serialization.StringSerializer.
```

3）value.serializer：指定用于序列化消息中 value 的程序。它的取值是序列化程序的完整类名。控制台生产者的默认取值如下。

```
org.apache.kafka.common.serialization.StringSerializer.
```

序列化是将对象转换为字节流，进而存储该对象或将其传输到内存、数据库或文件的过程。详情可参考 https://goo.gl/eDQE5A（https://docs.microsoft.com）。

基于 Kafka 库，可以轻松构建一个简单的生产者，具体步骤如下所示：

1）使用以下命令签出源代码库中的最新代码：

```
git clone
https://github.com/PacktPublishing/Data-Lakes-for-Enterprises
```

如果代码库已被复制到本地，使用以下命令搜索出最新的代码：

```
git pull
```

2）在代码库中，本章的源代码位于文件夹 chapter07 中。

3）以下是 chapter07/pom.xml 中声明的简单生产者的 maven 依赖。

```
<dependencies>
   <!-- https://mvnrepository.com/artifact/org.apache.kafka
   /kafka-clients -->
   <dependency>
       <groupId>org.apache.kafka</groupId>
       <artifactId>kafka-clients</artifactId>
       <version>0.10.1.1</version>
   </dependency>
</dependencies>
```

4）简单的生产者的 Java 实现如以下代码片段所示：

```
public class SimpleProducer {

   public static void main(String[] args) throws
   ExecutionException, InterruptedException {
```

```
        Properties props = new Properties();

       /*
       逗号分隔的broker列表，可能需要根据实际情况更新运行Kafka broker的VM IP
       */
        props.setProperty("bootstrap.servers",
        "192.168.0.117:9092");

        //设置处理消息key的serializer
        props.put("key.serializer",

"org.apache.kafka.common.serialization.StringSerializer");

        //设置处理消息(value)的serializer
        props.put("value.serializer",

"org.apache.kafka.common.serialization.StringSerializer");

        //创建一个生产者
        Producer<String, String> producer = new
KafkaProducer<String, String>(props);

        //创建一个消息用于发送至某个topic
        ProducerRecord message = new
        ProducerRecord("customer", "001", "A Sample Message...");

        //发送消息
        producer.send(message);

        System.out.println("Message Published");
       //关闭生产者连接
       producer.close();
   }
}
```

在前文的例子中，读者可能忽略了以下内容：

- Kafka 生产者所需的最小配置可以作为 properties 对象，也可以作为 map 对象。
- 使用默认的序列化程序的序列化消息的 key 和 value。
- 在发布消息时，需要指定 topic 和消息的 key。
- 指定的 topic 如果不存在，发送消息时会自动创建。

5）生产者要发布消息到远端的 broker 进程，需要按照图 7-14 所示设置 server.properties 文件中的配置，文件位置在 ${KAFKA_HOME}/config/server.properties，这样使得 Kafka 服务器可以绑定到正确的外部 IP 而不是 localhost。

6）上述源代码可以通过在读者的 IDE 中执行主程序来运行。唯一的先决条件是必须运行 Kafka。确保在运行该类之前改写 SimpleProducer 类中（第 18 行）的 broker IP。一旦消息发布到该 topic，控制台将输出已发布的声明消息，如图 7-15 所示。

7）可以运行以下命令从 Kafka 控制台验证已发布的消息：

```
${KAFKA_HOME}/bin/kafka-console-consumer.sh --topic customer --
bootstrap-server <broker-ip>:9092 --from-beginning
```

```
############################# Socket Server Settings #####################################

# java.net.InetAddress.getCanonicalHostName() if not configured.
#   FORMAT:
#     listeners = security_protocol://host_name:port
#   EXAMPLE:
#     listeners = PLAINTEXT://your.host.name:9092
listeners=PLAINTEXT://192.168.0.165:9092
```

图 7-14 Kafka Server Host（外部 IP）绑定配置

```
[centos@localhost kafka_2.11-0.10.1.1]$ bin/kafka-console-consumer.sh --topic
customer --bootstrap-server 192.168.0.117:9092 --from-beginning
A Sample Message...
```

图 7-15 console consumer 消费 SimpleProducer 发布的消息的输出

Kafka Connect

正如前面所讨论的，Kafka Connect 可以用来复制进出 Kafka 的流式信息，与 Flume 非常相似。然而，为了使用 Kafka Connect，需要下述的依赖来实现合流设置。Confluent 的安装步骤如下：

1）使用下面的命令下载最新的 Confluent 安装包。下载的内容较大，可能需要花些时间才能完成：

```
wget http://packages.confluent.io/archive/3.2/confluent-oss-3.2.0-2
.11.tar.gz
```

2）切换到用户目录，并使用以下命令提取前该 tar 包的内容。把提取的 Confluent 文件夹设置为 ${CONFLUENT_HOME}：

```
tar zxvf ${DOWNLOAD_DIRECTORY}/confluent-oss-3.2.0-2.11.tar.gz
```

3）使用以下命令将 CONFLUENT_HOME 设置为环境变量，并将其添加到 ~/ .bashrc 文件中：

```
export CONFLUENT_HOME=<PATH to CONFLUENT Directory>
```

4）使用以下命令启动 ZooKeeper 和 Kafka 服务：

```
${KAFKA_HOME}/bin/zookeeper-server-start.sh
${KAFKA_HOME}/config/zookeeper.properties
${KAFKA_HOME}/bin/kafka-server-start.sh -daemon
${KAFKA_HOME}/config/server.properties
```

5）使用下列命令启动格式注册器：

```
${CONFLUENT_HOME}/bin/schema-registry-start
${CONFLUENT_HOME}./etc/schema-registry/schema-registry.properties
```

一旦格式注册器启动成功，控制台会出现下列日志信息：

```
[2017-05-09 22:10:10,088] INFO Server started, listening for
requests...
(io.confluent.kafka.schemaregistry.rest.SchemaRegistryMain:45)
```

非卖品，仅供非商业用途或交流学习使用

6）按照以下 URL（从“快速入门指南”中的第 5 步开始）中提供的 Avro 格式示例的快速入门说明进行操作。“快速入门指南”中指定的所有命令都需要在 ${CONFLUENT_HOME} 目录中运行，并且所有的 localhost 都需要使用与 Kafka 绑定的 IP 进行替换（http://docs.confluent.io/2.0.0/quickstart.html#quickstart）。这是一篇很好的快速入门文章，可以让读者了解 Kafka Connect 的用途。

7）运行 Avro 示例后，切换回 ${KAFKA_HOME} 目录并运行以下命令查看测试 topic 的结果：

```
${KAFKA_HOME}/bin/kafka-console-consumer.sh --topic test --
bootstrap-server <broker-ip>:9092 --from-beginning
```

8）到这里，已经在已安装的 Kafka 中运行 ZooKeeper 和 Kafka。可以在 Confluent 平台中运行模式注册器（schema registry），并能够以 Avro 格式推送消息，以并行的方式消费这些带有 Avro 验证信息的消息。

7.11.3　消费者：从 Kafka 获取数据

Kafka 默认的消费者配置（$KAFKA_HOME/config/consumer.properties）信息如下：

```
# ZooKeeper 连接字符串
# 逗号分隔的host:port对，与ZooKeeper集群节点一一对应
# 例如"127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002"
zookeeper.connect=127.0.0.1:2181

# ZooKeeper连接超时时长，以ms为单位
zookeeper.connection.timeout.ms=6000
# 消费者群组 id
group.id=test-consumer-group

# 消费者超时时长
# consumer.timeout.ms=5000
```

由此可见，console consumer（默认使用的 Kafka 消费者，基于终端或 SSH）唯一需要的配置信息是 ZooKeeper connection（这是一个旧属性，为保持向下兼容而保留）或者 bootstrap server（这是一个新属性）及一些详细配置项和消费者群组 ID。其他关于超时和群组 ID 的配置是可选项。消费者群组 ID 表示消费者属于某个群组，使得该组对每个消息总是只处理一次。如前面生产者部分所述：消息的 key 和 value 的反序列化默认使用字符串反序列化器。

综上，消费者必须配置的信息包括：

- bootstrap.servers：指 Kafka broker 监听地址（以逗号分隔的 ip:port 格式），它表示 Kafka 集群中的 broker。
- key.serializer：指定用于序列化消息中 key 的程序。它的取值是序列化程序的完整类名。控制台生产者的默认取值是 org.apache.kafka.common.serialization.

非卖品，仅供非商业用途或交流学习使用

StringSerializer。

- value.deserializer：指定用于序列化消息中 value 的程序。它的取值是序列化程序的完整类名。控制台生产者的默认取值是 org.apache.kafka.common.serialization.StringSerializer。
- group.id：定义了一个消费者群组。

接下来将解释如何使用消费者 API 构建一个简单的消费者。

1）简单消费者程序的 maven 依赖关系与生产者相同，如下所示，请读者参考：

```
<dependencies>
   <!-- https://mvnrepository.com/artifact/org.apache.kafka/
   kafka-clients -->
   <dependency>
       <groupId>org.apache.kafka</groupId>
       <artifactId>kafka-clients</artifactId>
       <version>0.10.1.1</version>
   </dependency>
</dependencies>
```

2）一个简单消费者程序的 Java 实现如下所示：

```
public class SimpleConsumer {
   public static void main(String[] args) {
       Properties props = new Properties();

       /*
       设置 broker 地址列表，用逗号分隔。需要用实际运行
       Kafka brokder 的 VM IP 来替换它们
       */
        props.setProperty("bootstrap.servers",
        "192.168.0.117:9092");

        // 设置处理消息键 (key) 的 deserializer
        props.put("key.deserializer",

"org.apache.kafka.common.serialization.StringDeserializer");

        // 设置处理消息值 (value) 的 deserializer
        props.put("value.deserializer",

"org.apache.kafka.common.serialization.StringDeserializer");

        // 设置 gourpld
        props.put("group.id", "1234");

        // 创建一个 Kafka Consumer 类型的消费者
        Consumer<String, String> consumer = new
        KafkaConsumer<String, String>(props);

        // 为消费者订阅 topic
        consumer.subscribe(Arrays.asList("customer"));


        try {
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
            while (true) {
                // 获取最新 offset 后的所有记录
                ConsumerRecords<String, String> records =
                consumer.poll(100);

                // 显示所有记录
                for (ConsumerRecord<String, String> record :
                 records) {
                    System.out.println("key:" + record.key() +
                    "\nvalue:" + record.value());
                }
            }
        } finally {
            consumer.close();
        }

    }
}
```

从上述代码中，读者可以观察并验证以下内容：

- Kafka 消费者所需的最小配置可以作为 properties 对象，也可以作为 map 对象。
- 消费者不断检查 topic 中的新消息。
- 消费新消息的轮询时间间隔指定为 100ms，这个参数可以配置，如前边的代码所示。

3）读者可以自行选择 IDE，并执行主程序来运行前面的源代码。唯一的先决条件是 Kafka 必须在运行中，且生产者已经发布了一些消息。在运行该类之前，请确保 SimpleConsumer 类中的 broker IP 已更改。控制台输出将显示该 topic 中被消费的所有消息，如图 7-16 所示。

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java ...
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
key:001
value:A Sample Message...
```

图 7-16　简单消费者的输出

7.11.4　设置多 broker 集群

虽然在示例代码中设置的是单节点 Kafka 服务，但是随时可以在单节点上弹性地部署多 broker 服务。在这种情况下，如果一个 broker 发生故障，另外的 broker 仍然可以服务。

在单节点上部署多个 broker 也是非常简单的，需要在 ${KAFKA_HOME}/config/server.properties 中进行如表 7-2 所示的改动。

表 7-2　在单节点上部署多个 broker

配 置 参 数	含　义
broker.id	对于每个 broker 实例，这个 ID 在 Kafka 集群应该始终是唯一的
port	如果在单节点部署多个 broker，各个 broker 的端口必须设置成不同的
logs.dir	单个节点上的每个 broker 的日志位置必须定义为不同的路径

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

一般来说，建议针对每个 broker 实例配置 2 个不同的 server.properties 文件。该文件的示例如下所示（注意观察变更的参数）。2 个 broker 配置的 broker ID 是不同的。因此，需要配置日志文件的目录，以便 2 个 broker 写入日志到不同的文件夹中。

```
############################# Server Basics #############################

# broker的id，该id必须配置成一个整数，且不能与其他broker相同
broker.id=10

# 配置是否能删除topic，默认值是false，即不能删除
delete.topic.enable=true

############################# Socket Server Settings
#############################

# java.net.InetAddress.getCanonicalHostName() if not configured.
#   FORMAT:
#     listeners = security_protocol://host_name:port
#   EXAMPLE:
#     listeners = PLAINTEXT://your.host.name:9092
listeners=PLAINTEXT://192.168.0.117:9092

# broker会将自己的机器名和端口广播通知到producer和consumer。如果不设置，
  那么将使用listener的配置值，当然前提是listener被配置了。否则，将使用
  java.net.InetAddress.getCanonicalHostName()的返回值
#advertised.listeners=PLAINTEXT://your.host.name:9092

# 处理网络请求的线程数
num.network.threads=3

# 处理磁盘I/O操作的线程数
num.io.threads=8

# 套接字服务端使用的发送缓存（SO_SNDBUF）的大小
socket.send.buffer.bytes=102400

# 套接字服务端使用的接收缓存（SO_RCVBUF）的大小
socket.receive.buffer.bytes=102400

# 套接字服务端能接受的最大的请求大小（防止OOM）
socket.request.max.bytes=104857600


############################# Log Basics #############################

# 存储日志的文件位置，多个文件以逗号分隔
log.dirs=/tmp/kafka-broker10-logs

# 每个topic的默认分片数。分片数越多，计算的并行度就越高，但是也会导致更多的多broker
  存储的文件
num.partitions=1

# 每个数据目录用于在启动时恢复数据和在退出时存储数据的线程数量，如果数据目录存在于
  RAID阵列上，则推荐增大这个值
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
num.recovery.threads.per.data.dir=1

############################## Log Flush Policy
#########################

# 消息本来是会立即写入磁盘，但是默认情况下会通过fsync()方法实现系统缓存懒写入。
  下面是数据刷入磁盘的相关配置
# 这里是一些需要权衡的考虑：
#    1）持久化：如果没有副本，未刷入磁盘的数据可能丢失
#    2）延迟：太大的刷入间隔可能延迟峰值，因为可能在刷入磁盘时发现大量数据需要写入
#    3）吞吐量：数据刷入磁盘通常是开销极大的操作，较小的刷入间隔可能导致过多的寻址
#    下列配置项设定了数据刷入磁盘的策略：每过一段时间刷入一次、每N条数据输入一次，
    也可以两者兼顾。该配置可以是全局的，也可以针对各个topic单独配置
# 消息数量累积到下面的配置值就刷入磁盘
#log.flush.interval.messages=10000

# 消息刷入磁盘前在缓存中的最大等待时间
#log.flush.interval.ms=1000

############################## Log Retention Policy
##############################

# 下面是日志文件清理策略的相关配置。清理策略可以是设置一个过期时间或者数量累积到一定值。
  日志文件一旦满足任一条件就会被删除
# 符合删除条件文件的最小存在时间
log.retention.hours=168

# 基于大小的日志保存策略。一旦剩余的大小低于log.retention.bytes，就会从日志中删除段
#log.retention.bytes=1073741824

# 一个日志段文件的最大容量。一旦达到该值，就会创建一个新的日志段文件
log.segment.bytes=1073741824

# 检查日志段的时间间隔，查看根据保留策略是否需要删除日志段
log.retention.check.interval.ms=300000

############################## Zookeeper ##############################

# zookeeper的连接配置（详见zookeeper日志）。由逗号分隔的“机器名：端口”格式，每一项
  对应一台zookeeper服务器，例如，"127.0.0.1: 3000, 127.0.0.1: 3001, 127.0.0.1: 3002"。
  用户也可以在url后追加根目录，来指定Kafka的所有znodes的根目录
zookeeper.connect=localhost:2181

# 连接到zookeeper的超时时间，单位是毫秒
zookeeper.connection.timeout.ms=6000
```

完成上述更改后，用以下命令以守护进程模式启动 Kafka 服务器：

```
${KAFKA_HOME}/bin/kafka-server-start.sh -daemon
${KAFKA_HOME}/config/server-broker10.properties ${KAFKA_HOME}/bin/kafka-
server-start.sh -daemon ${KAFKA_HOME}/config/server-broker20.properties
```

在 Kafka 实例启动后，观察在 tmp 文件夹中创建的日志文件（/tmp/kafka-logs/），如图 7-17 所示。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
drwxrwxr-x.  2 centos   centos    152 Feb 16 19:43 kafka-broker10-logs
drwxrwxr-x.  2 centos   centos     92 Feb 16 19:44 kafka-broker20-logs
```

图 7-17 多 broker 日志

7.11.5 SCV 用例中的 Kafka 应用

图 7-18 清晰地概括了 Apache Kafka 在 SCV 项目中的应用。

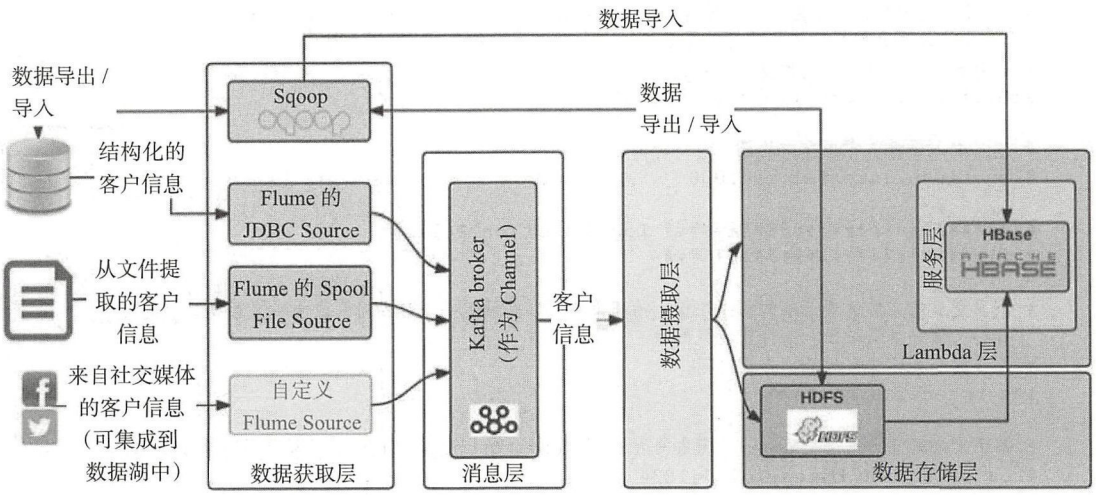


图 7-18 SCV 用例中的 Kafka

本章介绍了信息发布、消息服务器和消息消费等方面。在第 6 章中，我们使用 Kafka 作为 Flume 中的 Channel 和 Sink。在 Kafka 用作 Channel 时，它既是生产者又是消费者；当使用 Kafka 作为 Sink 时，它更多地承担了生产者的功能。

这基本上意味着，我们打算将 Kafka 更多地用作消息 broker 服务器和 Channel，以便在 SCV 中定义获取和消费接口。这些信息可以是结构化和非结构化信息的混合体，可以通过 XML/JSON 等标准的数据格式的消息实现信息交换。第 6 章介绍了如何获取结构化数据和非结构化数据，并作为脱机文件输入 Kafka 中。可以使用自定义序列化器 /Sink 构建附加或自定义接口，把 Kafka 用作消息 / 事件的中央 broker 服务器，通过数据摄取层将它们传输到目标系统中去。

7.12 适合使用 Kafka 的场景

Kafka 的核心功能使其适用于本书的用例，遇到以下情形时读者应该考虑使用 Kafka：

- 需要一个高度分布式的消息系统。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

- 需要一个可以指数级扩展的消息系统。
- 需要高吞吐量的发布 / 订阅功能。
- 需要系统能兼容不同能力的消费者，这些消费者可以订阅发布到主题中的消息。
- 需要容错操作时。
- 需要消息传递具有持久性。
- 显然，上述所有场景都不能接受性能下滑，因此它必然是高性能的。

7.13　不合适使用 Kafka 的场景

在一些场景和用例中，读者不应该使用 Kafka：

- 如果需要消息能够按顺序处理，则必须只能有一个消费者和一个分区。但是，这不是 Kafka 典型的工作方式，我们确实有多个消费者和多个分区（按照设计，一个消费者从一个分区中消费），因此，Kafka 不满足这种用例的期望。
- 如果需要实现任务队列，Kafka 也不适用，原因同上。Kafka 没有典型消息队列那么完备的功能。
- 如果需要一个典型的主题功能（先进先出），它的功能实现是完全不同的。
- 如果开发或生产环节基于 Windows 或 Node.js（尽管有些主观），读者需要知道在这些环境中使用 Kafka 多少会有一些问题。
- 如果需要更高的安全性和控制，Kafka 的最初设计没有真正基于安全考虑，这使 Kafka 的应用有时会受到影响。

7.14　其他选项

本章中详细介绍了 Kafka 的优点，也详细介绍了 Kafka 的一些不足以及不适合使用 Kafka 的场景。这意味着 Kafka 是一个适合本书以及 SCV 用例的选择，其他场合需要视需求而定。选择 Kafka 的主要原因在于 Kafka 明显的优势，特别是在处理大数据及其相关技术时。

市场上还有其他选择，它们是功能完善的消息中间件（MOM），与 Kafka 相比功能更丰富。本节简要总结了一些读者可以考虑用来代替 Kafka 的备选方案。不是说这些选项不能应用于本书的用例，而是我们认为 Kafka 是最合适的方案。如果要寻找其他方案来代替 Kafka，这些替代方案是最优先的选项。

所有的技术选型都经过了仔细的技术分析，本书希望在正确的方向上为读者提供正确的指导。

7.14.1　RabbitMQ

其中一个强有力的替代方案是 RabbitMQ。它是业界最为强大和知名的消息中间件之

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

一。RabbitMQ 的一些重要特性如下：

- 用 Erlang 语言编写。
- 文档质量高、在线资源丰富。
- 由 Pivotal 开发和维护。
- 软件自身开源并由众多供应商提供商业支持。
- 成熟。
- 丰富的路由功能。
- 天然分布式架构。
- 具有持久性和持久化选项（可选）。
- 根据需要，可以按照顺序处理消息。
- 根据需要，支持集群化运行。
- 基于 MPL 许可证开源。

以下场景可以选择使用 RabbitMQ：

- 需要较为复杂的消息路由机制。
- 保证消息的可靠传输。
- 不强制要求消息有序传输，由消费者自行维护消息的顺序。
- 企业的生产环境需要付费的商业支持服务。
- 部署方面具有高可用性。
- 需要良好的路由功能。
- 需要支持多种协议（AMQP、STOMP、MQTT 等）。
- 需要更强大的安全特性。
- 需要支持事务。

AMQP（Advanced Message Queuing Protocol）是应用层协议的一个开放标准，为面向消息的中间件设计。AMQP 的特性是面向消息、队列、路由（包括点对点和发布和订阅），可靠性和安全性。

——维基百科

STOMP（Simple（或 Streaming）Text Oriented Message Protocol，以前称为 TTMP）是一种简单的基于文本的协议，专为面向消息的中间件（MOM）而设计。它提供了可互操作的有线格式（wire format），允许 STOMP 客户端与任何支持该协议的消息代理进行通信。

——维基百科

MQTT（Message Queuing Telemetry Transport）是一种 ISO 标准，基于发布－订阅的“轻量级”消息协议，层次位于 TCP/IP 之上。设计它的目的是与需要小型数据或网络带宽

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

有限的远程位置连接。

——维基百科

7.14.2　ZeroMQ

ZeroMQ 的重要特性如下：

- 轻量级消息系统。
- 低延迟高吞吐。
- 能够在 ZeroMQ 中引入各种组件，可以处理许多消息传输场景。
- 可以进行异步编程。
- 对消息传输系统中的事务支持效果不佳。
- 基于 GPL 的许可证。
- 支持多种语言的连接。

以下场景可以选择使用 ZeroMQ：

- 需要实现最简单的消息系统。
- 需要消息能够快速传输。
- 需要消息架构能连接多个应用程序，并且节点上的空间开销很小。
- 需要异步消息传递模型。

7.14.3　Apache ActiveMQ

Apache ActiveMQ 的重要特性如下：

- 提供多种持久化机制。
- 水平可扩展。
- 配置高度灵活。
- 支持多种传输协议。
- 多个项目在内部使用 Apache ActiveMQ 作为企业服务总线。
- 支持许多高级消息系统的特性。
- 易于与许多应用程序集成，尤其是 Java 开发的应用程序。
- 可靠。
- 快速。
- 消息层支持事务。
- 开源，社区活跃。
- 支持调度器。

以下场景可以选择使用 ActiveMQ：

- 需要该产品易于配置。

仅供非商业用途或交流学习使用

- 需要高性能的消息系统，不需要持久化。

7.15 总结

本章涵盖的主题非常丰富。此时读者可以放心，因为已经对 Apache Kafka 有了充分的了解，足以使用 Kafka 来构建数据湖。

本章从数据湖中消息传输的相关内容入手，深入、详细地介绍了 Kafka，包括它的架构及其各个组件。之后，详细向读者展示了 Kafka 的完整工作实例，逐步介绍了从安装到使用 Kafka 实现端到端的消息传输的全过程。最后，与其他章节类似，我们介绍了其他可以代替 Kafka 的具有相同能力的技术方案。

通过阅读本章，读者应该对消息层有了清晰的认识，同时对 Apache Kafka 及其工作原理有了深入的了解。读者也应该清楚地了解在本书的用例中如何使用这项技术以及它的职能。

第 8 章 Chapter 8

使用 Apache Flink 处理数据

到现在为止，笔者可以确认读者已经熟知本书各章的表述策略了。本章内容沿用相同的策略。首先，将会介绍数据摄取层（Data Ingestion Layer），然后介绍技术路线，在我们的案例中，使用了 Apache Flink。

对于数据湖实现来说，它必须具备一个特征：能同时适当地处理流式数据和批量数据，Flink 是一种好选择。在本章中，我们将对 Flink 进行足够详细的介绍以确保读者能利用这些知识搭建数据湖示例系统。对 Flink 进行全方位的介绍超出了本书的范围，有此类需求的读者应寻求一本专门的 Flink 著作。本章首先深入介绍 Flink 的核心优势及缺点，然后介绍它的架构及重要组件，最后深入介绍 Flink 实操细节（含代码范例），以及 SCV 用例中的 Flink。

本章结尾介绍了除 Flink 之外的其他选择，当 Flink 不能满足读者所在企业中的需求时，可以考虑使用它们。

8.1 数据湖背景中的数据摄取层

本章将着手介绍数据湖的一个核心功能层级——数据摄取层。如果企业中存在需要处理来自不同应用程序的流式数据和批量数据的情况，那么该层级至关重要。

在这里能胜任前面所述的数据处理任务的框架就是 Flink，做出该选择的过程是艰难的，因为存在一些类似的技术框架，如 Apache Spark，这个框架也足够强大，并且更成熟。最后，基于 Flink 的某些特性我们选择了它。尽管如此，考虑到 Spark 在数据处理领域的显著特点，在本章结尾部分的“其他选项”节中会用一定篇幅介绍 Spark（本书其他章不再单

独介绍 Spark）。

本章将带领读者探究数据湖的数据摄取层，先介绍其工作原理，然后再介绍具体的处理框架，即 Flink。

8.1.1 数据摄取层

数据摄取指的是获取或导入数据用于中间处理或存储到数据库的过程。摄取指的是提取或吸收。

——whatis.com

在第 2 章中，读者应该已经对数据摄取层有所了解了。该层负责同时收集流式数据及批量数据，然后在这些数据上执行任意所需的处理流程。图 8-1 通过更好的图形化视图来帮助读者回忆该层级的功能：

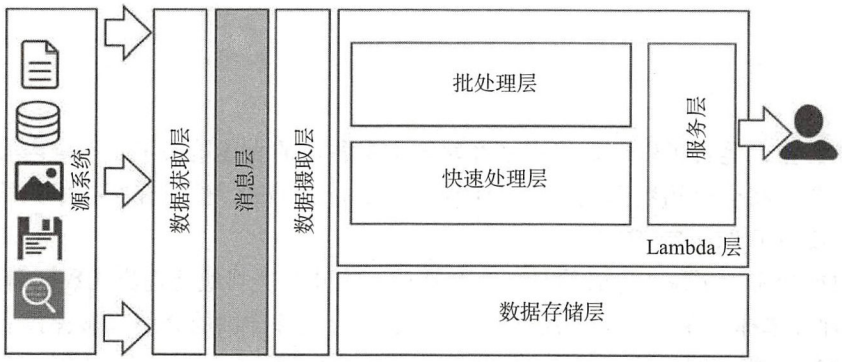


图 8-1 数据湖——数据摄取层

在我们的数据湖实现中，数据摄取层负责消费消息层传递过来的消息，完成所需的数据转换操作，从消息中提取所需的数据然后传递给 Lambda 层（批处理层及快速处理层），因此数据摄取层的输出格式需要迎合下一步存储或处理的要求。数据摄取层必须保证消息消费速度不低于消息产生速度，这样消息 / 事件处理就不存在延迟了。

下面列举了数据摄取层的一些特性：

- 能以简单、快捷的方式处理输入的数据（我们的案例中，输入层为消息层）。
- 能处理多种不同的数据流（实时流式数据或批量数据，同步或异步）。
- 能处理多种数据结构（结构化数据、半结构化数据、非结构化数据）。
- 集成了多种持久化存储机制。
- 支持多种传输协议。
- 支持大数据的 4V 特性。
- 能与多种不同的系统或技术连接。

如图 8-2 所示，该层从消息层读取数据，在传输给 Lambda 层（批处理层与快速处理层）

之前对数据进行相应的增强和转换。

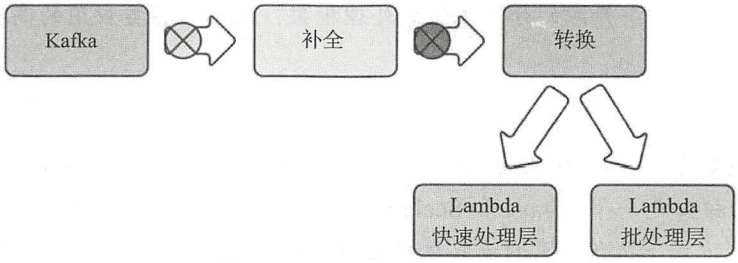


图 8-2　数据湖实现中数据摄取层的工作原理

8.1.2　数据摄取层——技术路线图

在我们的范例中，构建数据湖时采用 Flink 作为数据摄取层的技术框架。其他同类型的框架也不乏强有力的竞争者，如 Apache Spark，在笔者看来它是与 Flink 等同的好选择，本章后面部分将对它做一些介绍。本章主要向读者介绍 Flink。

图 8-3 从技术角度来描述数据湖的概念架构图，这种方法也会贯穿本书始终。我们首先介绍每一种技术框架，以及它与总体架构的关系，最后才会介绍如何整合这些框架（本书的第三部分）。

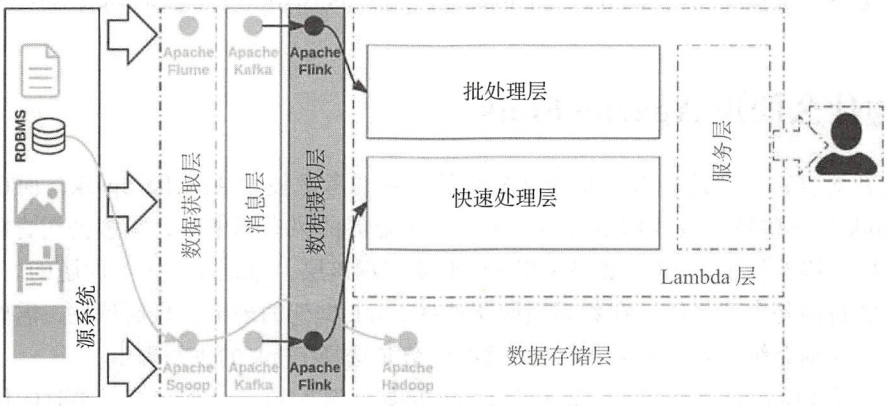


图 8-3　数据摄取层技术路线图

在我们的 SCV 用例中，来自消息层的数据被数据摄取层读取，在传输给 Lambda 层之前对数据进行相应的增强和转换，当然，也可能将增强或转换后的数据持久化到数据存储层中去。

该层中也可以使用 Kafka Consumer、Flume 等技术框架。为了在某种程度上展示 SCV 的真实运行范例，在本书的第三部分中将会整合这些技术，这样可以为企业数据湖应用展示出一个清晰的 SCV。

8.1.3 什么是 Apache Flink

Apache Flink 是一个开源的分布式流式处理框架，能满足各类应用的高吞吐、高可用、精确的数据处理要求。

——flink.apache.org

Apache Flink 是一个社区驱动的开源大数据分析框架，与 Hadoop 和 Spark 类似。Apache Flink 的核心是一个由 Java 和 Scala 语言开发的分布式流式数据处理引擎。Flink 可在并行和流水线方式下执行任意的流式数据处理程序。Flink 的流水线运行时系统允许执行批处理和流式处理程序。

——维基百科

Apache 官方的定义比较容易理解，维基百科上的定义则有点晦涩了。对读者而言，理解 Flink 为用户提供了一个处理流式数据及批量数据的统一编程模型即可。

本章并不打算向读者对 Flink 的工作原理做一个巨细靡遗的全方位介绍，那样的介绍需要一整本专门的著作才能胜任。

尽管本章不会对 Flink 过于细致地讲解，但仍会挑选其若干重要的方面进行介绍。通过对这些核心知识点的介绍，确保读者基于这些知识利用 Flink 搭建数据湖的相应模块。

如果想对 Flink 的使用有一个完全的了解，建议读者阅读 Apache Flink 最新的官方文档，请参考下面链接中的内容 https://ci.apache.org/projects/flink/flinkdocs-release-1.1/。

8.2 为什么使用 Apache Flink

对笔者而言，做出这个选择是非常艰难的。最初考虑使用 Apache Spark，而在撰写本书时，Flink 的一些特性让我们不得不三思，从工业界的风向标来看，很多公司青睐 Flink，这使得我们最终选择了 Flink。如果采用 Spark 来实现该层，也是完全没有问题的。

本节的目标是向读者陈述在数据摄取层选择 Flink 框架的理由，显而易见，需要一个单独的小节来详细介绍 Flink 的各种优势，这些优势是选择 Flink 的主要理由。

但在讨论 Flink 的利弊之前，先看看 Flink 是如何诞生的，以及它在开始时的优势是什么。它的很多特性来自现有的类似技术，博采众人之长这本身就是一种优势。另外，如果有新的需求，也会开发新特性来满足需求（有句著名的谚语是这么说的：需求是发明之母）。借鉴现有技术与实现新特性的主要区别在于实施层面（设计与细节的实现）。

下面列举的是 Flink 相对其竞争对手的一些主要优势：

- 抽象层次较高且简单易用的 API。
- 依靠 Flink 的多项内置特性进行快速的轻量级数据处理（内存中处理）。
- 能够接入每个流式数据，并在其之上执行所需的分析。

- 低延迟的数据处理。
- 支持精准的一次性处理（exactly once processing）。
- 高吞吐。
- 容错。
- 易于配置。
- 开源。
- 对延迟抵达数据或乱序数据流仍能提供精确的处理结果。
- 天然是有状态的。

Spark 是以微批量（micro-batch）方式处理数据的，因此我们选择 Flink 而不是它。微批量方式在处理实时数据时会存在问题（瓶颈），因为需要及时地按照用例的要求提供计算结果。仅仅由于这个原因就不得不放弃 Spark，因为微批量处理方式在分析实时数据或产生相关结果时必然会带来延迟。

8.2.1　Flink 简史

Flink 最早在 2014 年 4 月从 Apache 孵化器的 Stratosphere 项目孵化出来，并且在同年 12 月晋级为 Apache 的旗舰项目，Flink0.9 版是其首个旗舰项目。2016 年 12 月发布的 1.1 版是最新版本。

2010 年，研究项目“Stratosphere：Information Management on the Cloud”（由德国研究基金会（DFG）资助）由柏林科技大学、洪堡特大学、柏林大学和波茨坦研究所等机构合作启动。

——维基百科

图 8-4 通过图形化方式以用户容易理解的方式描述了 Flink 的演化过程。

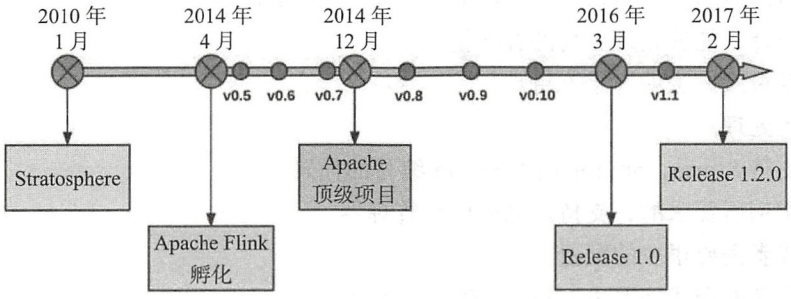


图 8-4　Apache Flink 简史

Stratosphere 0.2 演进到 Flink 0.8 时系统进行了彻底重写。Flink 中增加了对实时数据流的支持，并能整合各种前端、后端技术，这使得 Flink 在工业界非常受欢迎。同时它也集成

了多种开源大数据技术，与大数据生态系统黏合得很好。从那时起，Flink 社区一直在蓬勃发展，并开始挑战像 Spark 这样的成熟框架，因为它的精巧实现避免了微批处理，能真正对流式数据进行实时处理。

8.2.2 Flink 的优势

Apache Flink 因其内在的优点及新颖的设计理念而被广大用户选择。考虑到我们处理的案例及数据量的变化，以及如何帮助企业通过使用流式数据来创造价值，我们选择了 Flink。微批处理天然存在延迟，Flink 这方面做得很棒，能做到低延迟处理实时数据，这更能被现代工业界接受。

下面列举的是 Flink 的核心优势：

- 除了 HDFS，也支持其他类型的文件系统。
- API 简单易用。
- 开源。
- 久经考验的功能和技术支持。
- 极快的数据处理速度。
- 为批处理及流式数据处理提供了统一的编程模型，这有助于企业安排技术资源（人员）来处理数据湖中的各种数据。
- 可以对流式数据进行极其快速的分析和转换。
- Flink 拥有自主的机器学习库（machine learning library），即 FlinkML，用于处理机器学习相关案例。
- 原生支持迭代式查询及算法执行。
- 支持许多新的操作符以及内置的 MapReduce 模型。
- 基于流水线引擎（pipelined engine）的低延迟数据处理。
- 通过设计，能够处理更高的吞吐量。
- 使用轻量级方法的容错支持。
- 支持批处理。
- 内置支持 Storm，允许重用为 Storm 编写的代码。
- 根据用例所要求的，支持定制的内存管理器。
- 无须或者需要很少的配置。
- 大多数性能调优问题由 Flink 引擎自己完成。
- 对于现代大数据技术所需的流式数据，具有不同的窗口特性（用于处理稳定的数据流，Flink 根据时间戳、计数和其他标准将到达的数据划分为片，这称为窗口）。
- 有状态的特性使 Flink 避免陷入失效的泥潭。
- 能够大规模地在企业部署，能够处理来自不同业务线应用程序的高容量的快速数据。

- 将 API 与实际执行解耦（这意味着同一个程序可以以多种方式运行，对用户屏蔽了执行细节），这样有助于获得更好的性能。
- 通过 REST API 对外提供监控和管理所必需的多种统计指标。
- 提供易于使用的仪表盘（dashboard）来丰富用户交互体验。
- 支持高可用的集群设置。

想了解更多细节，可阅读 Flink 官方文档，参考以下链接：http://flink.apache.org/introduction.html#features-why-flink。

8.2.3　Flink 的劣势

作为这一领域最新的（严格来说不是最新的，它的起源可以追溯到 2008 年）框架，Flink 确实试图完善它的竞争对手所拥有，同时自身也存在的许多缺点。所以在那个领域中，目前来看它的劣势更少。

下面列举的是 Flink 的一些劣势：

- 与竞争对手相比，在写作本书时，Flink 的受欢迎程度和社区接纳程度都不高。
- 行业的成熟程度较低。
- Flink 的流水线执行在内存管理（对于长时间运行的流水线）和容错方面有一定的限制。
- Flink 使用原始字节数组作为内部数据表示，如果需要，很难通过编程来操作它们。
- 没有成熟的用于查询数据的 API（Flink 的 Table API 与其他竞争对手相比并不是很好）。
- 数据源集成（Data Source Integration）API 不是最好的，使用时存在很多限制。
- 目前只有 Java 和 Scala API 可用。

8.3　Flink 的工作原理

一图胜千言，因此本节由一张图片开始，图 8-5 描绘了 Flink 的运行原理。

Flink 能够同时处理批量数据和流式数据。它将批量数据当成一种特殊的流式数据来处理，这本身就是 Flink 的一个独特特性。我们在第一部分中对 Kappa 架构做了一些解释，其中所有数据都被当成流式数据来处理，Flink 在其架构和实现中遵循了该原则。

在图 8-5 中，来自各种源系统的两类数据（批量数据和流式数据）都涌入了 Flink。Flink 程序提交作业（job），主节点（master）与工作节点（woker）各司其职处理数据，最终产生输出。

8.3.1　Flink 架构

Flink 架构的关键是 3 个重要组件的协同工作。

- Client（客户端）
- Job Manager（作业管理器）

- Task Manager（任务管理器）

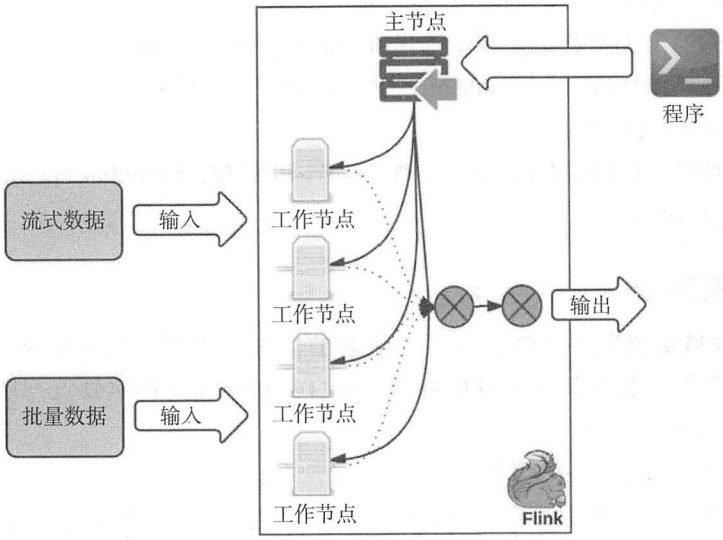


图 8-5 Flink 的运行原理

为了便于理解，图 8-6 中将三者的关系抽象出来。

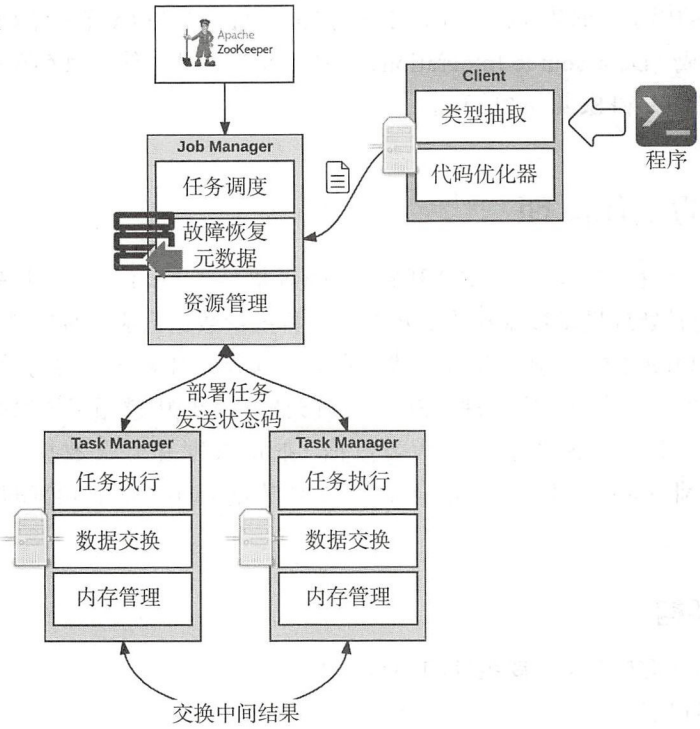


图 8-6 Flink 架构中的核心组件

图 8-6 非常清楚地展示了三大核心组件之间的交互关系，下面将对这些组件逐一深入介绍。

1. Client

Client 提供了下列功能，供程序执行时使用：

- 代码解析（Code Parsing）。
- 类型抽取（Type Extraction）。
- 代码优化器（Optimizer）：Flink 有内置的代码优化器用于在程序执行前对代码进行优化，产生最优的可执行代码。
- 图构建器（Graph Builder）：构建每个作业中数据流的关系图。
- 将构建好的数据流关系图传输至 Job Manager。
- 取回作业的输出结果。

图 8-7 剖析了 Flink Client 的组件构成以及与其他 Flink 组件的关系。

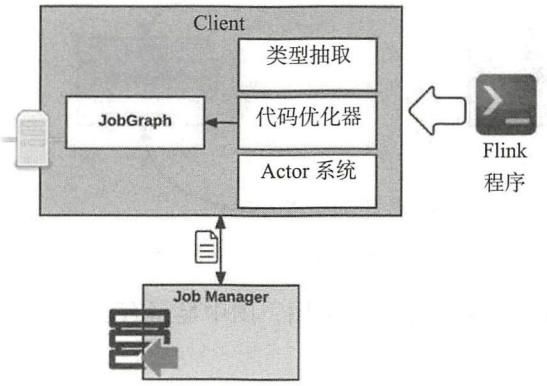


图 8-7　Flink Client 组件构成

2. Job Manager

Job Manager（位于 Master 节点上）的主要功能如下：

- 为并行执行 Task 做准备。
- 任务调度：调度 Task Manager 上的任务，它可以部署、停止、撤销 Task Manager 上的任务。
- 包含 checkpoint coordinator，用于触发 Task Manager 上的 checkpoint 操作。
- 任务执行追踪：追踪 Task Manager 中任务的执行情况以及收集执行结果。
- 收集、维护与数据流关系图相关的元数据。

图 8-8 展示了 Job Manager 的组件构成情况，以及它与 Task Manager 的关系。

3. Task Manager

Task Manager 为工作节点（worker node），任务在这里真正执行。它由多个任务槽（task slot）构成，根据 Job Manager 指令要求，任务在任务槽中执行。

仅供非商业用途或交流学习使用

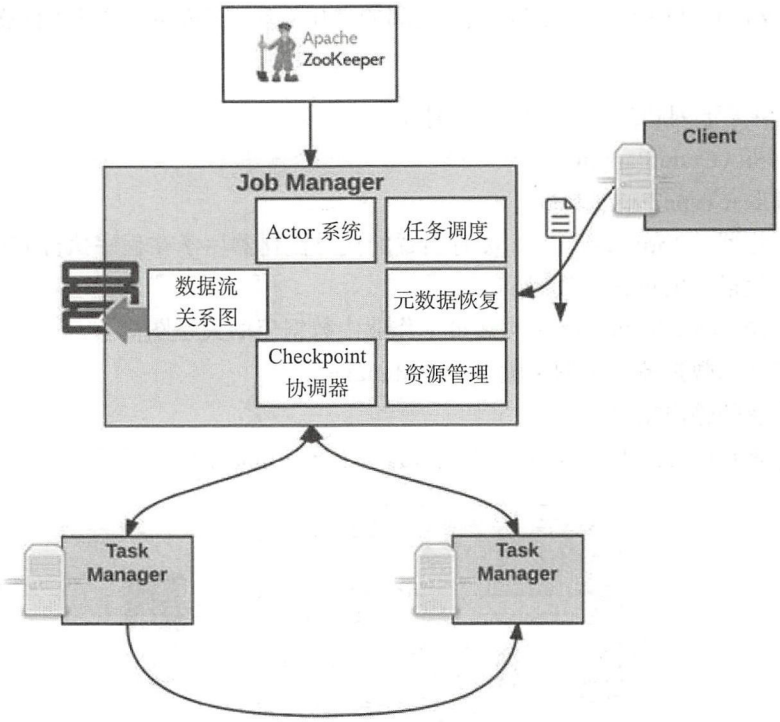


图 8-8 Job Manager 组件构成

图 8-9 所示是 Task Manager 组件构成图，图中显示有多个工作节点，实际生产部署中通常会有不止一个工作节点。

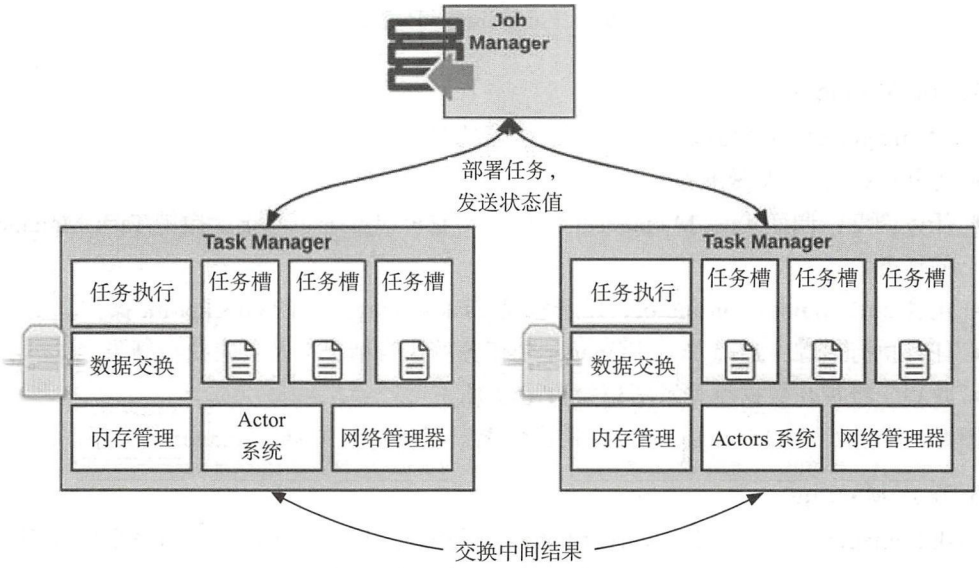


图 8-9 Task Manager 组件构成

仅供非商业用途或交流学习使用

图 8-10 显示了 Flink 架构中的所有组件，但是与之前的高层次抽象不同，它详细描述了每个组件及其交互关系。

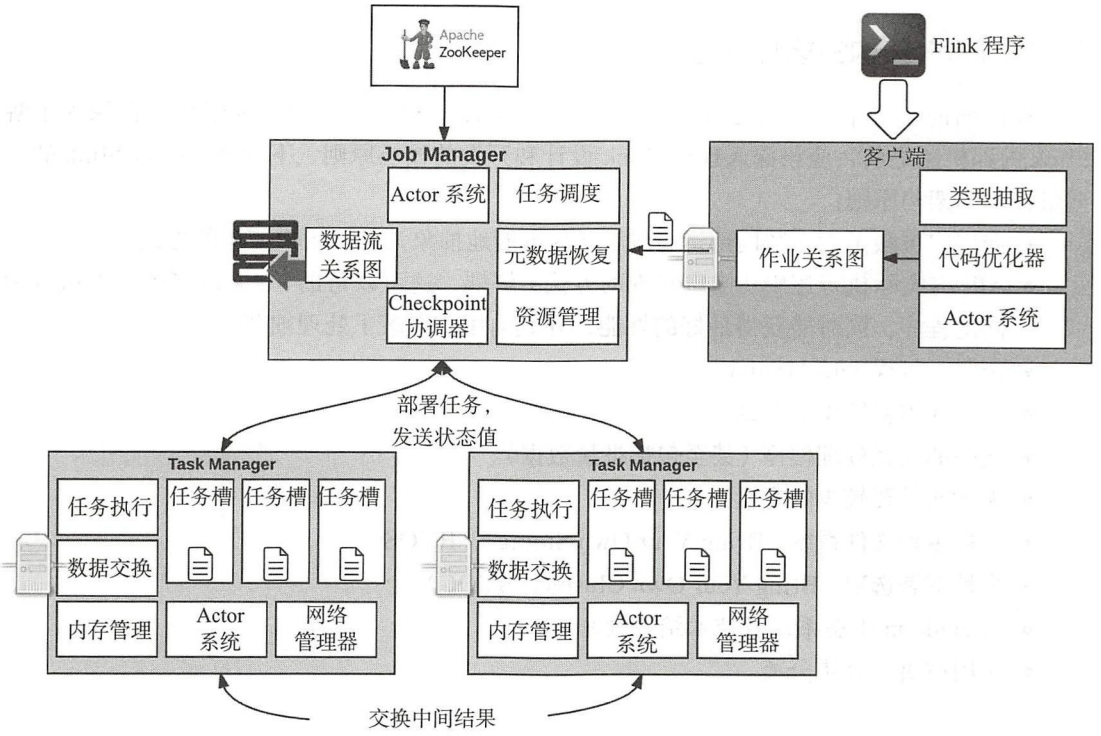


图 8-10　Flink 架构详情（参考了 Flink 官方文档）

前面的图中显示，每个组件都是由一个基于 Akka 库的协调系统连接的。Flink 有不同的部署拓扑。如果 Flink 是在本地模式下启动的，那么 Job Manager 与它的一个或多个 Task Manager 将在同一个 JVM 中启动。

如前面的图所示，Client 向 Job Manager 提交作业，而 Job Manager 又控制 Task Manager，其中任务被放置在任务槽中，然后执行。

下面给出了 Flink 执行模型以及作业为获取计算结果而执行的计算路径。

图 8-11 揭示了 Flink 的执行模型。Flink 程序首先被编译器解析，然后进行类型抽取，之后对生成的代码进行优化（Flink Optimizer）。每个被提交的作业转化为数

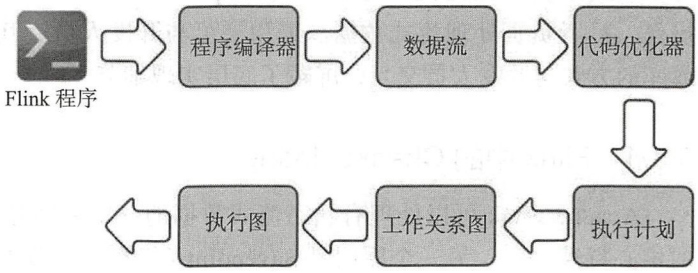


图 8-11　Flink 执行模型

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

据流关系图，最后传输给 Job Manager，在这里会创建一个执行方案，作业关系图会被传输给 Task Manager，任务最终在 Task Manager 中执行。

8.3.2 Flink 架构的核心原则

在前面的小节中，我们已经了解了 Flink 的整体架构以及它的重要组件。在深入了解 Flink 内部机制之前，应该深入理解 Flink 设计和开发的核心原则。下面列举的是 Flink 的一些最核心的架构原则：

- 向用户屏蔽了大多数内部运行细节（尽可能地抽象为易于用户使用的方式）。
- API 与实际执行逻辑（接口实现的方法）解耦，这可以帮助 Flink 以多种方式执行相同的程序，同时能获得最好的性能，并且向用户屏蔽了处理细节。
- 声明式编程（declarative）。
- 无须或需要较少的调试。
- 统一的流式处理模型（甚至包括批量数据）。
- 无须或需要较少的配置。
- 支持多种文件系统（Bring Your Own Storage，BYOS）。
- 多种部署选项（Bring Your Own Cluster，BYOC）。
- 与 Hadoop 生态系统及技术结合紧密。
- 应用逻辑与容错分离。

8.3.3 Flink 组件栈

图 8-12 描述了 Flink 的组件栈（component stack），对组件栈中每个组件做深入全面的介绍超出了本书的范围，有兴趣的读者可以阅读 Flink 官方文档以进一步了解（https://ci.apache.org/projects/flink/flink-docs-release-1.1/）。

图 8-12 以分层模式展示了 Flink 组件栈中的各种组件（概念架构）。如图所示，Flink 支持多种存储系统（BYOS），存储系统组成最下面的一层。紧接着的一层中是 Flink 支持的部署技术（BYOC），读者可以在图中看到，它支持多种部署技术。接下来的一层中，Runtime 是 Flink 执行模型的核心，它由 Batch Optimizer 和 Stream Builder 组成，前者负责处理批量操作，后者负责处理流式数据。最顶层可与开发人员交互，该层做了良好的抽象，以 API 或库的方式与开发人员交互，屏蔽了底层实现细节。

8.3.4 Flink 中的 Checkpointing

Apache Flink 可以处理有状态流式数据的一个原因是 Checkpointing，这是它与众不同的核心特性之一。另一个原因是 Savepoint（将在下一节中解释），这也使得 Flink 能够处理有状态流式数据。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

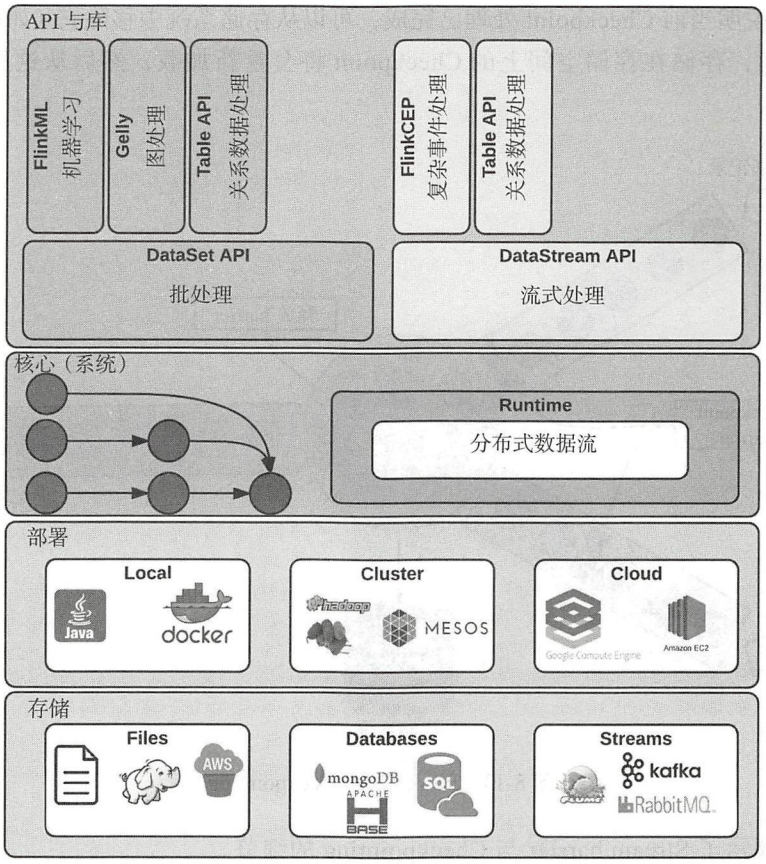


图 8-12　Flink 组件栈（Flink 官方文档—— http://flink.apache.org）

容错能力是 Flink 的核心特性之一，在获取高吞吐、高性能的同时还能做到容错，这是非常具有挑战性的，而 Flink 通过 Checkpointing 特性做到了这一点。

与批量数据（定义了起点和终点）不同，流式数据没有明确的起点和终点。此外，流入的流式数据必须保留状态信息，这为实现容错带来了额外的挑战。

Flink 的 Checkpointing 使用了分布式快照（distributed snapshot）方法，该方法基于 Chandy 和 Lamport 在 1985 年提出的一种技术。该算法在 Flink 中略有变化，在处理流式数据时，后台定期保存快照状态（频率是可配置的），它支持持久化存储（Flink 支持多个文件系统）。但是，这种机制并没有给其他组件带来任何压力，非常轻量级，并且对开发人员来说是透明的。

Flink 的 Checkpointing 工作机制借助了 Stream barrier 机制。这些 barrier 在数据源被插入，然后在数据流中流动。这些流如图 8-13 所示，它们是当前快照或下一个快照的一部分。当这些 barrier 通过运算符时，会触发状态快照处理，并进行持久化存储。一旦 Checkpointing 完成，barrier 流过运算符，处理流程继续。当所有接收器（Sink）接收了该

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

Checkpoint，说明当前 Checkpoint 处理已完成，可以从存储系统中移除它（如果需要的话）。如果发生故障，存储在存储空间上的 Checkpoint 将会重新加载，然后从这里恢复并继续处理。

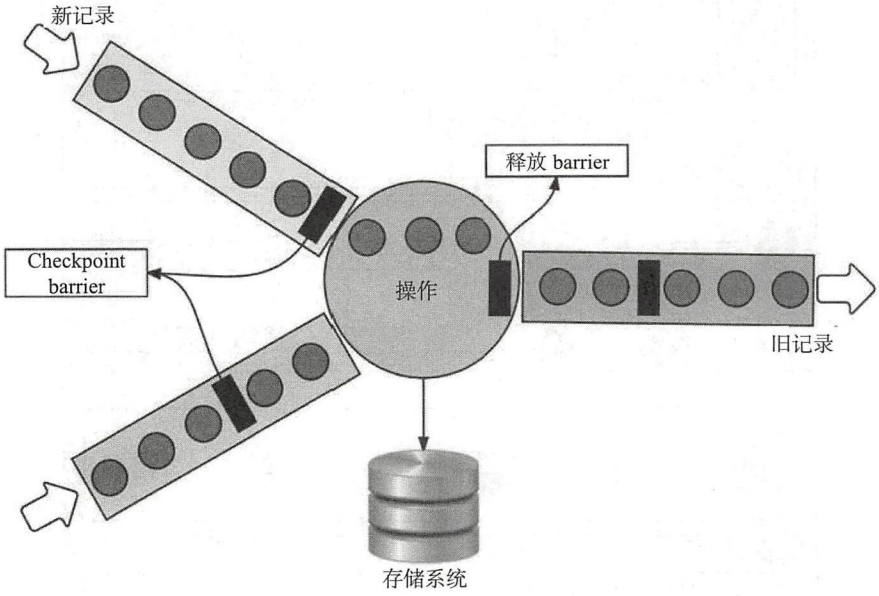


图 8-13 Flink 中的 Checkpointing

图 8-13 展示了 Stream barrier 与 Checkpointing 的细节。

Checkpointing 是 Flink 中用到的非常重要的一种技术，其他的很多设计原则，如容错，完全依赖这项技术。

8.3.5 Flink 中的 Savepoint

Savepoint 是 Flink 的另一个重要特性，这也是它领先诸多竞争对手的重要原因。Savepoint 是某个时间点的快照，它记录了我们在输入流的处理过程中的精准位置，并保存了相关的元数据。它还可以跟踪所有挂起（pending）状态或在 Flink 执行引擎中正在执行的会话。从概念上来说，它就像停止流数据来拍照一样。然而，这样做并不能真正停止操作，而是在后台默默地运行。

在 Flink 内部，Savepoint 与 Checkpointing 的处理方式比较类似，但是它们有一些值得注意的差别，如下所列：

- 可以通过相关配置来触发。
- 除非用户干预，否则不会被 Flink 终止。

在图 8-14 中，通过图形化的便于理解的方式向用户介绍 Savepoint 的工作原理。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

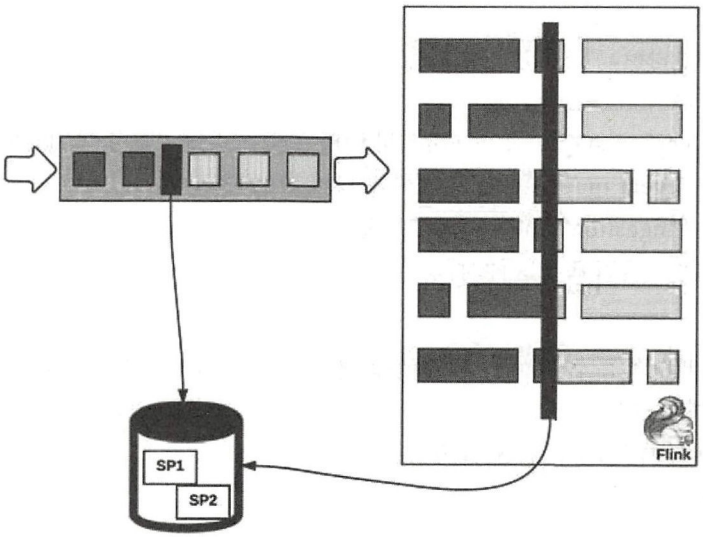


图 8-14　Flink 中的 Savepoint（SP）

Savepoint 功能可以让我们轻而易举地实现某些操作，同时也提供了一个内部的快照版本机制。如下所列：

- 使 Flink 升级 / 修复 bug 变得容易。如果由于任何原因导致了问题，它总是可以使用以前的 Savepoint 来恢复。
- 可以帮助进行 A / B 测试。Savepoint 可以用来将 Flink 中不同版本代码与测试的执行分离开。
- 可基于某些原因对数据流重新进行处理，或者在某些特定场景下进行重放（replay）。

想了解更多 Savepoint 的细节，建议阅读下面这篇博客：http://data-artisans.com/how-apache-flink-enables-new-streaming-applications/。

8.3.6　Flink 中的流窗口选项

一般来说，窗口定义了无边界数据流中一个包含有限元素的子集。窗口定义可以基于时间、元素计数、计数与时间加权或某些自定义的逻辑。

——Flink 官方文档（flink.apache.org）

处理无限的数据流需要相应的窗口功能。Flink 的 DataStream API（将在后面的章节中讨论）确实有一些内置的窗口功能，能满足大多数用例的需求。它还允许我们根据用例的需要自定义窗口行为，让开发人员实现其接口并实现适当的方法。

下面是 Flink 内置的窗口选项：

- 时间窗口
- 计数窗口

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

1. 时间窗口

顾名思义，时间窗口按照时间来对数据流中的输入数据进行分组。Flink 中的时间戳有以下类型：

- 处理时间（processing time）：Flink 进程所在操作系统 / 服务器的时间。
- 事件时间（event time）：数据流事件本身携带的时间戳。
- 摄取时间（ingestion time）：前两个选项的折中。当每个事件到达时，记录事件到达系统的时间，然后基于这种时间对事件进行分组。

2. 计数窗口

顾名思义，此时按配置好的计数阈值来划分窗口大小。当事件计数达到阈值，相应的事件被归入同一窗口 / 组。

计数窗口或时间窗口都两种配置方式：

- Tumbling
- Sliding

3. Tumbling 窗口配置

Tumbling 窗口顾名思义，即滚动窗口。窗口之间不重叠，在滚动窗口中，事件不能同时属于两个窗口。可以为时间窗口或计数窗口进行 Tumbling 配置。

图 8-15 演示了 Tumbling 窗口的工作原理。

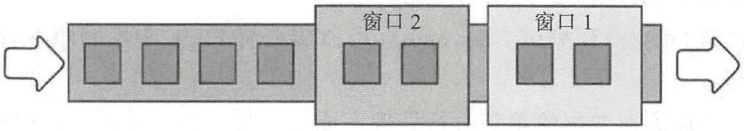


图 8-15 Tumbling 窗口

4. Sliding 窗口配置

顾名思义，窗口在数据流上滑动。由于这种滑动性质，窗口往往与其他滑动窗口重叠。因此，一个窗口中的数据流事件也可以属于其他窗口。同样，可以为时间窗口和计数窗口进行 Sliding 配置。

图 8-16 演示了 Sliding 窗口的工作原理。

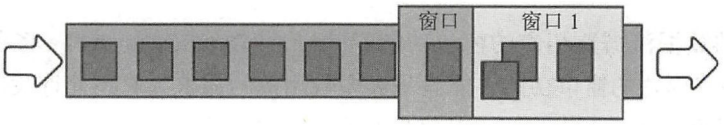


图 8-16 Sliding 窗口

在这一小节中，我们只是浏览了 Flink 的重要功能之一的流窗口，尤其是在处理无限数据流时，这是必备的功能。当需要进行聚合时，流窗口变得更为重要。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

8.3.7　内存管理

Flink 的内存管理被认为是开创性的，Flink 是最早实现了自定义内存管理的大数据技术之一。为了实现内存管理的目的，Flink 实现了自己的类型抽取（type extraction）及序列化（serialization）机制。

图 8-17 中描述的是 JVM 堆内存，它被划分为若干个区域，每个区域处理特定的内存活动。Flink 有一个托管堆，这是一种专门的内存分片，与 Flink 的相关操作有关，如缓存和数据处理。

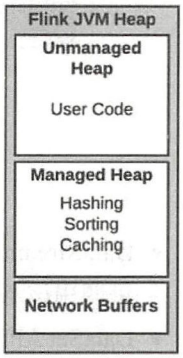


图 8-17　Flink 内存管理

下面是 Flink 内存管理的一些核心特性：

- 无 GC（垃圾收集）的策略。
- 序列化基于单个字段而不是整个对象。
- 使用 C++ 风格的内存管理而不是 Java（GC）风格的。
- 使用了内存池，内存池由多个页（page）构成，每个页由字节数组和 Map 对象组成。

以上列举的内存管理特性为 Flink 带来了许多独特的优势，如下所列：

- 不抛出内存溢出异常（Out of Memory，OOM）。
- 垃圾收集对硬件来说是一种压力，因为 Flink GC 的这些特性不会带来压力。
- 二进制形式的数据存储更高效，更节省内存。
- 几乎不需要内存方面的调优。
- 性能稳定可依赖。
- 不同二进制表示之间可以很容易地进行比较和操作，因而有更好的性能。

在计算机科学中，垃圾收集是自动内存管理的一种形式。垃圾收集器，或称收集器，试图回收垃圾或程序不再使用的对象占用的内存。1959 年左右，John McCarthy 发明了垃圾收集，以将 Lisp 中的手动内存管理抽象剥离出来。

——维基百科

对垃圾收集的全面介绍超出本书的研究范围，更多信息可参考以下链接：https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=53741525。

8.4　Flink 的 API

Flink 中的基本操作可以用非常简单的术语来表述，如图 8-18 所示。

图 8-18 中，Flink 程序连接到数据源，然后执行业务逻辑相关操作（变换、计算等），最后将结果输出到接收端（Sink）。Flink 可以处理两种形式的数据：实时数据（流式数据）

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

和批量数据（数据集）。相应地，提供了两类 API。

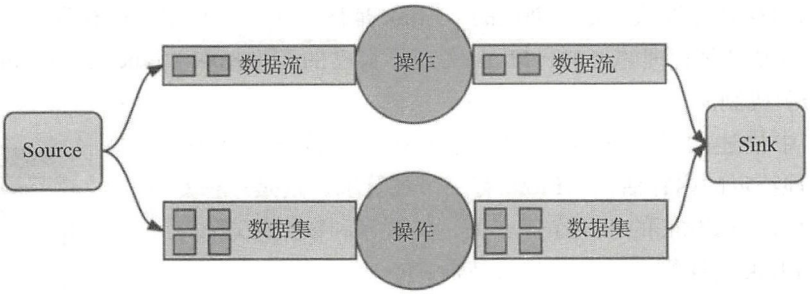


图 8-18 Flink 程序执行

- DataStream API：对实时数据流进行处理，适用于实时流分析，Java 及 Scala 语言都可调用该 API。
- DataSet API：用于处理费时的批量数据，Java、Scala 及 Python 等语言可调用该 API。

除了这些主要的 API 之外，Flink 还提供了很多领域相关的库，图 8-19 介绍了一些主要的领域相关库，后续将会详细介绍它们。

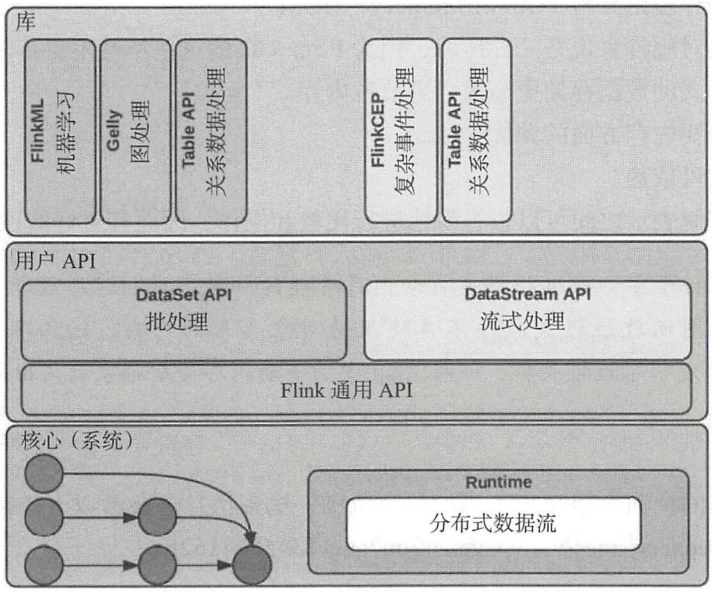


图 8-19 Flink API

8.4.1 DataStream API

顾名思义，DataStream API 在 Flink 中可以对流入的数据流做任何操作。操作对象可能是流中的某一个元素，也可能是流窗口中的元素集合。它提供了许多内置的数据转换操作，

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

并且提供了一个工具箱，用户可以根据业务需要，实现自定义的数据转换操作。DataStream 是 DataStream API 的核心结构。

Flink 的 DataStream API 几乎能处理任意类型的数据（基础数据类型如字符串，复杂类型如元组、POJO 等）。

图 8-20 揭示了 Flink 中 DataStream 程序的执行过程。

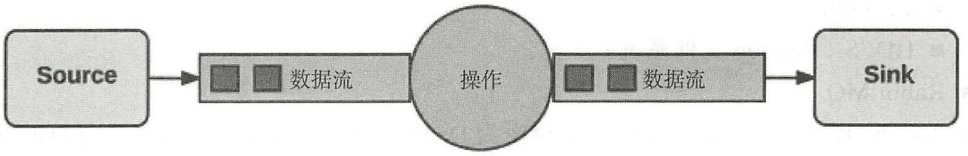


图 8-20　Flink 中 DataStream 程序的执行过程

1. Flink DataStream API 范例

为了处理数据流中的输入数据，编程时需要处理以下几个步骤（可参考 Flink 官方文档）：

1）获得 StreamExecutionEnvironment 对象。

StreamExecutionEnvironment 是所有 Flink 数据流执行程序的基础。下面的代码演示了如何在 Java 中获取该类型对象：

```
final StreamExecutionEnvironment env = StreamExecutionEnvironment
. getExecutionEnvironment( );
```

2）从适当的数据源加载初始数据。

3）对加载后的数据做必要的转换。所有可用的数据转换操作可参考以下 Flink 文档：https://goo.gl/YI82xY。

4）确定数据转换后结果数据的存储目的地。

5）触发程序执行。

这里我们并不想机械地复制 Flink 官方文档，只是建议读者对 DataStream API 做深入了解，可参考下面的文档：https://goo.gl/NS69SK。

2. Streaming connector

Flink 提供了多种 connector，便于开发者集成各种数据源，下面是连接的详细分类。

1）内置 connector：此类 connector 随 Flink 一起安装，由多个提供者开发、支持及维护。内置 connector 可分为以下几类：

- File：从指定的文件路径读取文件。它可以监视一个目录，当添加或更改文件时可以读取和执行所需的文件。
- Collection：从元素中读取数据，也可以从 Java Collection 中读取数据。
- Socket：读取指定端口 Socket 中的文本数据。
- Basic
  - File

非卖品，仅供非商业用途或交流学习使用

- Socket
- Standard output（标准输出）

- Advanced
  - ElasticSearch：用于存储和索引 JSON 格式的文档，后面有专门的章节详细介绍这门技术。Flink 同时支持 ElasticSearch1.x 及 ElasticSearch 2.x。
  - Cassandra
  - HDFS（Hadoop 文件系统）
- RabbitMQ
- Apache Kafka：可将 Kafka topic 转换为 DataStream，Kafka 也可以直接向 DataStream 写入数据。Flink 对集成 Kafka 提供了良好的支持。
- Apache NiFi
- Amazon Kinesis Streams
- Sources：Flink 中被配置为数据源的 connector。
- Sinks：Flink 中被配置为数据接收端的 connector。
- Source and Sink：Flink 中既可以当成数据源也可以当成数据接收端的 connector。

2）自定义 connector：如果内置 connector 不能满足要求，Flink 也提供了自定义 connector 的机制，允许开发者实现自己的 connector。Flink 提供了对应的工具包，可帮助使用者轻松实现自定义 connector。使用该工具包（特定的接口），可实现自定义的源或接收端。

8.4.2 DataSet API

Flink 使用内置的 DataSet API 处理批量数据。DataSet API 还具有必备的对常见用例进行数据转换的机制。在进行转换后，一个 DataSet 被转化为另一个 DataSet。Flink 用 Java、Scala、Python 等编程语言提供了 DataSet API。在 Flink 中，批量数据的处理与流式数据的处理方式相同。图 8-21 显示了使用 DataSet API 处理批量数据的基本工作原理，这里使用 MapReduce 范式产生数据集（在数据读取和处理时创建多种类型的数据对象）。

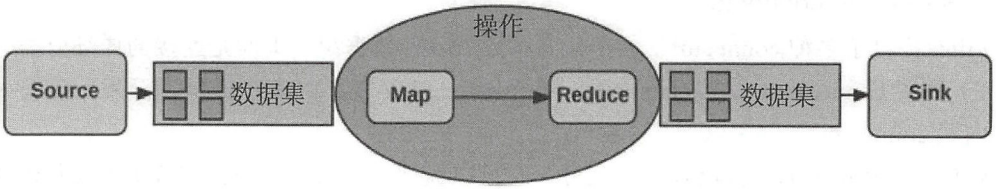


图 8-21 Flink 中 DataSet API 程序的执行过程

1. Flink DataSet API 范例

除了环境设置以外，DataSet API 程序的处理步骤与 DataStream API 相同。以下是 Flink 中使用 DataSet API 处理批量数据的真实步骤：

1）在使用 DataSet API 之前，需要获得一个用于处理批量数据的特定环境。在使用 API 之前需要获取 ExecutionEnvironment 对象，该对象的功能如下面这个代码片段所示（有很多类似的方法，这只是其中一个基本的方法）：

```
final ExecutionEnvironment env =
ExecutionEnvironment.getExecutionEnvironment();
```

2）要对数据进行操作，必须从数据源创建或加载数据。这些数据集有多种加载方式，可从文件加载，也可以从集合（collection-based）加载，还可以从 socket 以及其他来源加载。

3）下一步，对已加载 / 创建的数据做必要的转换。有多种方式对批量数据进行转换，如 Map、FlatMap、Filter、Reduce 等。Flink 文档中数据转换方式的完整列表可在下面的网址 https://goo.gl/HLg3bN 中找到。

4）在这个步骤中，指定转换后的数据结果存储位置（根据 Sink 配置）。例如，可能会将结果写入文件、Socket 等。

5）这是触发程序执行的最后一步，它触发实际的程序执行。

2. Table API

基于 DataStream API 和 DataSet API 构建的 API 可实现类似 SQL 的数据查询功能，该 API 称为 Table API。Flink 提供了 Java 和 Scala 语言版的 Table API。使用这些 API，可使用类 SQL 表达式来指定所需的操作。

Table API 在 DataStream 及 DataSet 之上进行了抽象，这使代码更易于编写。使用该方法，很容易通过使用流行的 SQL 表达式来处理结构化数据。API 处理的基本数据结构是表。

SQL（结构化查询语言）是一种特定领域的语言，可用于编程和设计，也可用于管理关系数据库管理系统（RDBMS）中的数据，还可用于 RDBMS 中的流式处理。

——维基百科

与 DataStream API 和 DataSet API 类似，使用 Table API 可以通过下列步骤进行：

1）与执行任意其他 Flink API 类似，需先获取 ExecutionEnvironment 对象：

```
ExecutionEnvironment env =
ExecutionEnvironment.getExecutionEnvironment();
```

2）之后，需要基于 ExecutionEnvironment 对象来创建 TableExecutionEnvironment 对象，可参考下面的代码。根据待处理的数据类型（批量数据或流式数据），需构建得到 BatchTableEnvironment（批量数据）或 StreamTableEnvironment（流式数据），如下面的代码片段所示：

```
BatchTableEnvironment tableEnv =
TableEnvironment.getTableEnvironment(env);
        OR
```

```
StreamTableEnvironment tableEnv =
TableEnvironment.getTableEnvironment(env);
```

3）现在，可以注册 table 了。可以基于 DataStream 或 DataSet 构建 table，如下面这个代码片段所示：

```
tableEnv.registerDataSet("table_name", mytable);
        OR
tableEnv.registerDataStream("table_name", mytable);
```

4）现在，使用 Table API，执行各种操作。

Flink 关于该 API 所有细节的文档可参考 https://goo.gl/B6F8EY。

8.4.3 Flink 领域相关库

除了核心 API（DataStream 和 DataSet）之外，还有其他 API 可用于帮助使用者快速、便捷地处理某些操作，如下面的列表所示。与 Table API 类似，大多数库是基于 Flink 核心 API 构建的：

- Gelly（Graph 库）
- FlinkML（机器学习库）
- FlinkCEP（复杂事件处理库）

事件处理是对与已发生的事件相关的数据（信息）流进行跟踪和分析（处理）并从中得出结论的方法。复杂事件处理（CEP）是一种事件处理，它将来自多个源的数据组合起来，以推断出更复杂情况的事件或模式。

——维基百科

1. Gelly——Flink 图数据库 API

Gelly 是 Flink 用于处理图数据库的 API 和库，如图 8-22 所示。该库有很多内置算法，用于提供图处理相关功能。图数据库管理功能最初被剥离到另一个库，即 Apache Spargel。然而一个新的项目，即由 Gelly 发起的 Gelly 项目又启动了图数据库研究，它与其他核心 API 步调一致，使用了 Flink 的核心功能。

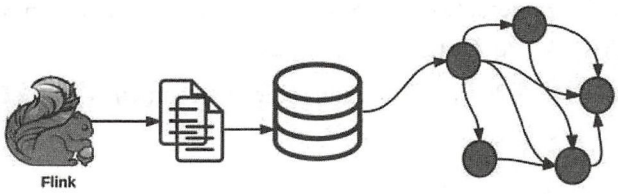


图 8-22 Flink 的图数据库 API（Gelly）

Gelly 库需要用到 DataSet 的派生类对象，因此它能满足批量数据的图处理需求。

Gelly 提供了各种图分析工具，也可以在图数据结构之上做多次迭代处理，同时 Gelly 也提供了许多处理图数据结构的算法。

与 Flink 的核心 API 类似，Gelly 提供了各种可对批量数据进行转换和修改的函数。Flink 中的图是一种由顶点（vertice）和边（edge）构成的数据结构（DataSet 对应的是批量数据）。

本节只是概述了 Gelly 库的功能，更多的细节可以参考网址 https://goo.gl/s5plEP 中的 Flink 官方文档。

Flink Gelly 提供了使用 Java 与 Scala 编程语言编写的 API。

2. FlinkML

Flink 提供了机器学习（ML）相关库，即 FlinkML。它内置了对各种机器学习库的支持，支持列表还在不断增长。它借鉴了 scikit-learn 的流水线风格的处理机制（http://scikit-learn.org/）。

FlinkML 既可处理批量数据也可处理流式数据。流式数据处理及 ML 应用非常重要，常见的案例有支付事务处理及检测欺诈（fraud detection）等。

更多的细节可参考 https://goo.gl/zDL2Cn 中的 Flink 文档。

Scikit-learn（更正式的名称是 scikits.learn）是一款针对 Python 语言的免费机器学习软件。

——维基百科

Apache SAMOA（Scalable Advanced Massive Online Analysis）是一款流式机器学习处理框架，可与 Flink 配套使用，以满足面向流式数据机器学习的应用需求。

3. FlinkCEP

Flink 有自己的复杂事件处理（Complex Event Processing，CEP）库，称为 FlinkCEP。该库允许用户对数据流中的事件模式进行处理来解决手头的问题。它基于 DataStream API 构建，帮助定义用户模式，然后注入数据流中，根据模式产生新的事件并进行相应的处理。

本书中并不打算深入探讨该主题，FlinkCEP 的全面介绍需要一整本书才能完成。强烈建议详细阅读 Flink 官方文档来了解相关知识：https://goo.gl/Tsu6q7。

8.5　Flink 运行范例

本节将详细介绍一个完整的 Flink 运行范例。最后会介绍案例中使用的 Flink 特性。

在撰写本书时，已经有 Flink 候选版本 3 的源代码可供下载，官方的 Flink 1.3.0 版本也即将发布。我们观察 FlinkRC3，它也被称为 1.3.0 版本，它预期实现的功能已经非常接近官方的 Flink 1.3.0 版本了。

8.5.1 安装

请按照下面的步骤对 Flink 进行完整安装：

1）下载最新的 Flink 源码，在撰写本书时，官方最新版本为 1.3.0，它与 RC3（发布候选版本 3）对应。在下载目录 ${DOWNLOAD_DIR} 中运行下面的命令：

```
wget https://github.com/apache/flink/archive/release-1.3.0-rc3.zip
```

2）切换到用户目录，并使用以下命令从 zip 文件中提取内容。将 Flink 源码文件夹解压到目标目录，${FLINK_SRC} 为源码目录：

```
unzip ${DOWNLOAD_DIR}/release-1.3.0-rc3.zip
```

3）切换至 ${FLINK_SRC} 目录，使用下面的命令来编译源代码。完整构建可能需要耗费一些时间。

```
mvn install -DskipTests
```

4）一旦构建完成，将生成的 build-target 文件夹复制到一个独立的用户目录 <flink-install-dir>。这将是 Flink1.3.0 RC3 的安装目录。下面的命令在执行过程中自动创建 Flink 安装目录：

```
cp -r ${FLINK_SRC}/build-target <flink-install-dir>
```

5）使用下面的命令配置环境变量 ${FLINK_HOME}，同时将它追加到 ~/.bashrc 文件中：

```
export FLINK_HOME=<flink-install-dir>
export PATH=$PATH:$FLINK_HOME/bin
```

6）切换至 Flink 安装目录，使用下面的命令启动 Flink 服务器：

```
${FLINK_HOME}/bin/start-local.sh
```

7）登录到 http://<vm-ip-address>:8081，可以看到类似图 8-23 所示的界面，在这里可以确定已经在单个节点上成功安装了 Flink。

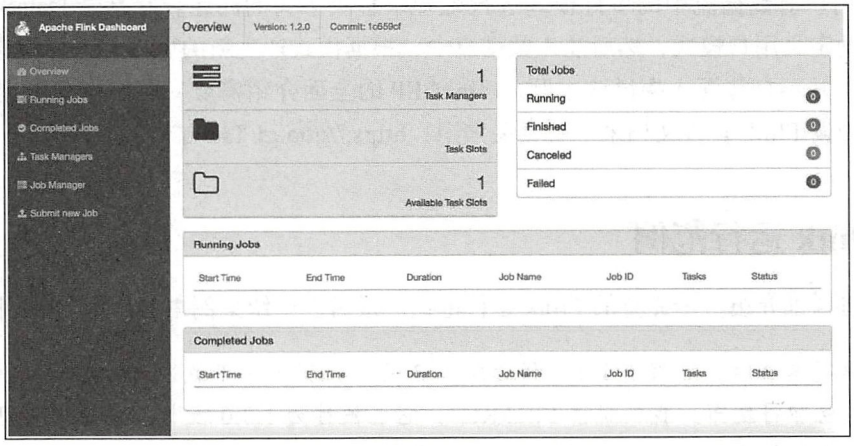


图 8-23 Flink 仪表盘（Dashboard）

如果打算在开发环境中运行 Flink 范例，明智的做法是在同一台机器上编译构建 Flink，因为类似的构建是依赖具体平台的。我们在本书撰写时使用了最新的 Flink 版本，即 RC3，此时大多数公共 Maven 仓库还没收录该版本。

8.5.2　范例——使用 Flink 处理数据

不妨来构建一个使用 Flink 处理数据的范例。为此，需先处理数据库中的已有数据，使之流经 Flink，最后将其存储在数据库中。

为了让范例看起来更真实，最好拥有较多的数据。在下面的小节中，我们将看到如何生成所需的数据。

数据生成

下面我们将分步骤说明如何生成所需的数据以演示范例：

1）范例中出现的这些命令或步骤，既可以运行在 IDE 中，也可以直接运行在 CentOS 命令行中。如果是在 CentOS 中执行，额外的 IDE 和 CentOS 之间的复制 / 移动步骤是可以避免的。

2）在本地 Git 仓库（克隆出来的），切换到包含本章范例代码的 chapter08 目录，我们将对它做详细介绍。本章范例数据生成相关的代码是 chapter08 中的一个模块，请参考 client-generator 子目录。

3）切换至 client-generator 目录，并运行以下命令基于源码构建项目：

```
mvn install
```

4）一旦构建成功，观察 client-generator 项目 target 目录下的 tarball 文件。

5）切换到某个用户目录，我们建议创建一个 ~ / data-generators 目录，然后利用下面的命令将 tarball 文件内容解压到该目录：

```
tar -zxvf <client-generator project>/target/client-generator-1.0-
SNAPSHOT-bin.tar.gz
```

6）一旦解压完毕，切换到对应的目录（~/data-generators/clientgenerator-1.0-SNAPSHOT/config/db.properties），基于你的数据库配置，对属性文件进行配置，下面是一个配置范例：

```
jdbc.url=jdbc:postgresql://<DB_IP_ADDRESS>/sourcedb?schema=public
user=
password=
```

7）为生成数据，请在 ~/data-generators/clientgenerator-1.0-SNAPSHOT 目录中运行下面的命令。为了演示我们的范例，一共生成了 100 000 条客户记录，记录中包括了他们的住址。请确保 PostgreSQL 数据库服务在执行以下命令之前已运行：

```
java -jar client-generator-1.0-SNAPSHOT.jar config/db.properties
100000
```

8）成功执行后，脚本将显示控制台中生成的记录状态。

此时数据库里已经有了超过 100 000 条的客户隐私数据（客户 + 地址），现在，我们来演练书中的范例，将 PostgreSQL 数据导出，使之成为数据流，经过 Flink 处理后，再存储到 Hadoop 中。

现在将以独特的方式构建示例，使之能够作为 SCV 用例的一个可扩展部分。

整个示例可以分为多个步骤，下面将会详细介绍。

1. 准备数据流

该步骤涉及数据流准备，具体来说是将数据发布到 Kafka 的 topic 中，这里将用到前面提及的简单的生产者 API（在第 7 章中解释过）。在这一步中，将从数据库中读取数据并将记录发布到 Kafka 的一个 topic 中：

1）为了准备数据流，请切换到 chapter08 目录，这里包含本章的范例代码。相关代码为 chapter08 目录中的一个模块，请参考 chapter08/flink-example1 子目录。

2）根据环境更改以下属性文件：

- chapter08/flink-example1/config/db.properties：该文件包含所有数据库相关的属性，用于访问数据库。
- chapter08/flink-example1/config/producer.properties：该文件包含所有跟生产者从数据库读取数据，发布客户记录到 Kafka topic 相关的配置。
- chapter08/flink-example1/config/flink.properties：此文件包含 Flink 的相关配置，目前只包含 HDFS 路径和用户账户。
- 范例程序中主要处理逻辑在 DBProducer.java 文件的类中，完整的代码可参考后面的代码片段。DBProducer 类从数据库中读取客户记录，然后将它们以 JSON 字符串形式发布到 Kafka topic 中去，topic 名为 customer。代码注释良好，非常便于理解。

```
public class DBProducer {

private static final String CUSTOMER_QUERY = "SELECT * FROM
CUSTOMER";

public static void main(String[] args) throws SQLException,
IOException,         ClassNotFoundException {
publishCustomers();
    }

private static void publishCustomers() throws SQLException,
IOException,         ClassNotFoundException {

//初始化 ObjectMapper 实例用于序列化 / 反序列化
        ObjectMapper mapper = new ObjectMapper();

//加载 producer 属性，初始化 Kafka 生产者
        Properties producerProps =
```

```
        PropertyLoader.loadProperty("producer.properties");
        Producer kafkaProducer = initializeProducer(producerProps);

//建立数据库连接和执行查询以检索全部客户记录

          Connection conn = getConnection();
          System.out.println("Database Connection Established...");
          Statement stmt = conn.createStatement();
          ResultSet result = stmt.executeQuery(CUSTOMER_QUERY);
          System.out.println("Query Executed...");

//将所有客户记录序列化为 JSON 串，并将它们发布到 Kafka topic 中

            while(result.next()) {
                Customer cust = new Customer();
                cust.setId(result.getInt("id"));
                cust.setFirstName(result.getString("first_Name"));
                cust.setLastName(result.getString("last_Name"));
                cust.setDob(result.getDate("dob"));

//将对象序列化为 JSON 串
              String customerMessage =
mapper.writeValueAsString(cust);

//将客户记录作为 JSON 消息发布
              ProducerRecord message = new
ProducerRecord(producerProps.getProperty("topicName"),

              String.valueOf(cust.getId()), customerMessage);
                kafkaProducer.send(message);
            }
            System.out.println("Messages Published");

//关闭生产者，程序退出
              conn.close();
              kafkaProducer.close();
            }

private static Producer initializeProducer(Properties
producerConfig) throws IOException {
                Producer<String, String> producer = new
KafkaProducer<String, String>(producerConfig);
return  producer;
            }

private static void closeProducer(Producer producer) {
if (producer != null) {
                    producer.close();
                }
            }

private static Connection getConnection() throws
ClassNotFoundException, IOException, SQLException {
                Class.forName("org.postgresql.Driver");
                Properties props =
PropertyLoader.loadProperty("db.properties");
return DriverManager.getConnection(props.getProperty("jdbc.url"),
```

```
props);
            }
        }
```

如前面的代码所示，程序从数据库中读取所有客户记录，然后将它们发布到一个名为customer 的 Kafka topic 中去。

此时需要将客户记录转换为对象，并且使用 ObjectMapper 将对象序列化为 JSON 形式。

3）在 chapter08 / flink-example1 目录中，运行下面的命令来编译代码：

```
mvn install
```

4）执行上面的代码，可以从 IDE 或命令行执行 DBProducer.java。在执行 Java 类之前，请浏览前面所列的那些类的代码，理解其工作原理。客户记录将以队列形式被发布到 Kafka topic 中：

- 在 IDE 中执行类，请确保设置工作目录指向正确的项目文件夹，即 flink-example1，在 Run Configuration 中设置模块 classpath。可以导航到顶层菜单 Run 中，然后在其中选择 Run... 打开这些设置。设置好之后，就可以从 IDE 中便捷地执行 Java 类了。这是关于 IntelliJ IDEA 的设置，Eclipse 中也有类似的设置。
- 如果是从命令行中执行类，可以在项目目录（chapter08 / flink-example1）中运行下面的命令：

```
java -cp target/flink-example1-1.0-SNAPSHOT.jar
com.laketravels.ch08.db.producer.DBProducer
```

2. 通过 Flink 消费数据流

现在，100 000 条记录以队列方式进入 Kafka，下一步就是使用 Flink 消费这些消息，此时会在 Flink 内部建立一个执行流水线。

Flink 有很多内置的 connector，Kafka connector 是 Source connector 的一种。为了引入该 connector，需要将以下依赖加入项目的 pom.xml 文件（flink-example1 /pom.xml）中：

```
<dependency>
   <groupId>org.apache.flink</groupId>
   <artifactId>flink-connector-kafka-0.10_2.10</artifactId
   <version>1.2.0</version>
</dependency>
<dependency>
   <groupId>org.apache.flink</groupId>
   <artifactId>flink-streaming-java_2.11</artifactId
   <version>1.2.0</version>
</dependency>
```

为了消费 Kafka topic 中的消息，可参考下面的代码。相同的代码在范例程序的 com.laketravels.ch08.consumer.FlinkProcessor 类中可以找到，这里包含了 main 函数：

```
final ObjectMapper mapper = new ObjectMapper();
Properties flinkProps = PropertyLoader.loadProperty("flink.properties");
```

```
//创建执行环境
StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

env.enableCheckpointing(2000, CheckpointingMode.EXACTLY_ONCE);

//解析用户参数
ParameterTool parameterTool = ParameterTool.fromArgs(args);

DataStream<Tuple2<IntWritable, Text>> messageStream = env.addSource(
new FlinkKafkaConsumer010(
parameterTool.getRequired("topic"),
new Tuple2DeserializerSchema(),
parameterTool.getProperties()));

messageStream.rebalance().print();
```

为了成功加载执行流水线，FlinkProcessor 类的 main 函数需要以下参数。本范例所需的参数有：

- topic：包含了待消费消息相关 topic 的名称。
- bootstrap.servers：包含由逗号分隔的，用 ip:port 标识的 Kafka broker 进程列表。
- zookeeper.connect：包含 ip:port 形式的 ZooKeeper 连接地址。
- group.id：确定消费者群组 ID，以及群组级别的消费者偏移量（offset）管理。

在我们的例子中，通过命令行参数来传递参数。一旦参数被传递，它们将被 ParameterTool 解析，并在代码中做必要的替换：

```
--topic customer --bootstrap.servers <KAFKA_SERVER_IP>:9092 --
zookeeper.connect <ZOOKEEPER_IP>:2181 --group.id 1 --auto.offset.reset
earliest
```

在上面的代码中，消息被 SimpleStringSchema 反序列化器消费。FlinkKafkaConsumer 需要使用它对数据流中的消息进行反序列化。

为了重放（replay）消息或重新提交 job，Flink job 可以用不同的群组 ID 来运行。

此时已经从 Kafka 中消费了消息，Flink 作为一个 Channel，将数据持久化到 HDFS 中去。细节将在下一步骤中解释。

3. 将数据流写入 HDFS

Flink 也提供了大量的 connector，其中包括 HDFS connector（作为 Sink）。所有的 HDFS connector 结构上是相似的。HDFS connector 可以从 Flink 的 DataStream 接收消息，消息结构为元组（tuple）类型。HDFS 也将数据存储为 tuple 类型。Flink 提供了 Tuple2 类来处理消息。

元组是有限的有序元素列表（https://en.wikipedia.org/wiki/Tuple）。

非卖品，仅供非商业用途或交流学习使用

任意 Sink 可通过调用 env.addSink（...）方法添加到 Flink 环境中去。范例中使用了 BucketingSink 类，下面的代码有助于理解我们的范例：

```
System.setProperty("HADOOP_USER_NAME", flinkProps.getProperty("hdfsUser"));
BucketingSink<Tuple2<IntWritable, Text>> hdfsSink = new
BucketingSink<Tuple2<IntWritable,
Text>>(flinkProps.getProperty("hdfsPath"));
hdfsSink.setBucketer(new DateTimeBucketer("yyyy-MM-dd--HHmm"));
hdfsSink.setWriter(new SequenceFileWriter<IntWritable, Text>());
hdfsSink.setBatchSize(1024 * 1024 * 400);
messageStream.addSink(hdfsSink);
```

如果试图将前面步骤中所示的代码与之前的代码结合起来，那么我们可能认识到一些与 Source 和 Sink 相关的问题。

这里的主要问题是，在前面的步骤中，Source 发送字符串形式的消息到 Kafka 中，而 Sink 需要处理元组结构的消息。它们之间数据格式的差异是明显的，需要做相应的处理。

为了解决这个问题，可以从数据源上实现一个自定义的反序列化模式类，参考下面的实现：

```
public class Tuple2DeserializerSchema implements DeserializationSchema {
   public Object deserialize(byte[] bytes) throws IOException {
      ObjectMapper mapper = new ObjectMapper();
      Customer cust = (Customer) mapper.readValue(new String(bytes),
         Customer.class);
      Tuple2<IntWritable, Text> tuple = new Tuple2<IntWritable, Text>();
      tuple.setFields( new IntWritable(cust.getId()), new Text(new
         String(bytes)));
      return tuple;
   }

   public boolean isEndOfStream(Object o) {
      return false;
   }

   public TypeInformation<Tuple2<IntWritable, Text>> getProducedType() {
      return new TupleTypeInfo<Tuple2<IntWritable, Text>>
         (TypeExtractor.createTypeInfo(IntWritable.class),
         TypeExtractor.createTypeInfo(Text.class));
   }
}
```

在 FlinkProcessor 类的 DataStream 初始化过程中，可以通过自定义序列化器（Tuple2DeserializerSchema）代替字符串序列化器，以下是完整的代码：

```
public class FlinkProcessor {

public static void main(String[] args) throws Exception {

final ObjectMapper mapper = new ObjectMapper();
        Properties flinkProps =
PropertyLoader.loadProperty("flink.properties");
// 创建执行环境
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
        StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

        env.enableCheckpointing(2000, CheckpointingMode.EXACTLY_ONCE);

// 解析用户参数
        ParameterTool parameterTool = ParameterTool.fromArgs(args);

        DataStream<Tuple2<IntWritable, Text>> messageStream =
env.addSource(
new FlinkKafkaConsumer010(
parameterTool.getRequired("topic"),
new Tuple2DeserializerSchema(),
parameterTool.getProperties()));

        messageStream.rebalance().print();

        System.setProperty("HADOOP_USER_NAME",
flinkProps.getProperty("hdfsUser"));
        BucketingSink<Tuple2<IntWritable, Text>> hdfsSink = new
BucketingSink<Tuple2<IntWritable,
Text>>(flinkProps.getProperty("hdfsPath"));
        hdfsSink.setBucketer(new DateTimeBucketer("yyyy-MM-dd--HHmm"));
        hdfsSink.setWriter(new SequenceFileWriter<IntWritable, Text>());
        hdfsSink.setBatchSize(1024 * 1024 * 400);
        messageStream.addSink(hdfsSink);

        env.execute();
    }
}
```

前面的例子可以在 IDE 中作为一个独立的进程运行或者从命令提示符运行。如果需要从命令行运行此命令，可使用下列命令，假定工作目录是本示例的源目录，即 flink-example1 目录：

```
java -cp target/flink-example1-1.0-SNAPSHOT.jar
com.laketravels.ch08.consumer.FlinkProcessor --topic customer --
bootstrap.servers <kafka-server-ip>:9092 --zookeeper.connect <zookeeper-
ip>:2181 --group.id 1 --auto.offset.reset earliest
```

请确保 HDFS 服务已运行并且在执行 FlinkProcessor 之前已对外开放可访问的 IP。如果 HDFS 被配置为运行在本地主机，则需要对 core-site.xml 做一定的修改。配置范例如下：

```
<property>
  <name>fs.defaultFS</name>
  <value>hdfs://192.168.0.165:9000</value>
</property>
```

这个执行的输出是 JSON 形式的客户信息，偕同客户 ID 存放在 HDFS 上，如图 8-24 所示。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

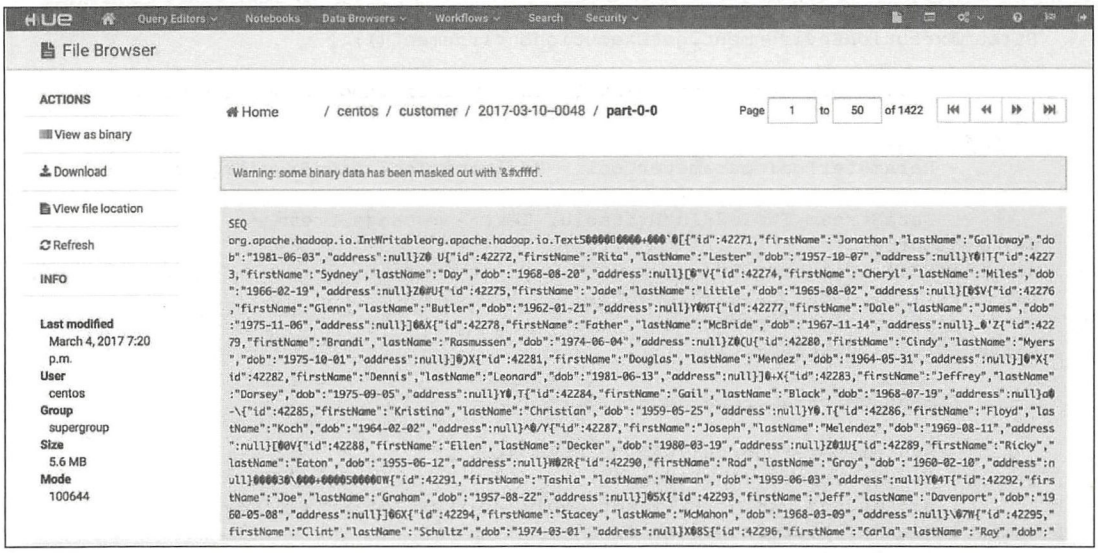


图 8-24 写入 HDFS 的消息

8.5.3 SCV 用例中的 Flink

现在，读者已经看到了基于 Flink 的数据处理的示例，是时候将其应用到 SCV 用例中，继续构建数据湖了。为此，可考虑集成存储在关系数据库和日志中的客户位置信息。对于用户日志，将它们生成 spool 文件，然后在 Flume 中通过 Kafka Channel 来消费它们。Kafka Channel 中的消息将被 Flink 处理流水线消费、处理，最终进入 HDFS Sink 中。在这个用例中，Flume 充当了数据获取层，从数据源获取数据，并将它们存储为 Kafka topic 中的消息。我们将定义两个 Flink 处理流水线，它们将从两个 Kafka topic 中消费这些信息，然后在 HDFS 层中处理和存储这些信息。

1. 用户日志数据生成

为了生成一个 customer location 的样本日志（包含 100 000 条记录），请按照以下步骤进行操作，这些步骤与前面的 customer 生成非常类似。

1）复制或更新项目源代码（假设该操作已完成，因为读者通常在最开始阅读本书时已经从 git 中复制了整本书的代码）。

```
chapter08/web-generator
```

2）切换至包含了项目的目录（web-generator）中，执行以下命令来生成 tarball 文件：

```
mvn install
```

3）切换至用户目录 ~/data-generators，使用下面的命令来解压文件：

```
tar -zxvf <web-generator project>/target/web-generator-1.0-
SNAPSHOT-bin.tar.gz
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

4）切换至文件解压后所在的用户目录，并运行以下命令以生成 spool 文件（location.log）形式的数据记录：

```
java -jar web-generator-1.0-SNAPSHOT.jar location.log 100000
```

5）生成的日志内容具有图 8-25 所示的结构。


```
"id":100095,"longitude":160.98048,"latitude":-62.541742
"id":100096,"longitude":157.16113,"latitude":68.544508
"id":100097,"longitude":11.078648,"latitude":82.648036
"id":100098,"longitude":-133.38267,"latitude":-10.155658
"id":100099,"longitude":-107.13184,"latitude":-88.434630
"id":100100,"longitude":-58.465704,"latitude":-44.031773
```


图 8-25　location 日志内容

正如我们在上面看到的，location 日志格式非常自由，这些日志可能包含其他字段。这里客户信息 id 可用于与客户信息关联。

6）将生成的 location.log 复制至 spool 目录下，目录将会在 Flume 配置文件中配置。该目录也是 Flume 范例运行时创建的目录。

2. Flume 设置

如果需要将数据库中数据或日志文件以数据流形式发布到 Kafka topic 中，则需要对 Flume 进行必要的配置。在这里将遵循与前几章中类似的步骤来配置数据源：

1）在 ${FLUME_HOME}/conf 目录中创建一个新的 Flume 配置文件。文件不妨命名为 customer-data-kafkaChannel-flume-conf.properties。顾名思义，这里包含了捕获客户数据所需的所有配置。

2）现在让我们看看数据源的配置，如下所示：

```
agent.sources = sql-source spool-source

agent.sources.spool-source.type=spooldir
agent.sources.spool-source.spoolDir=<directory containing the
spool file>
agent.sources.spool-source.inputCharset=ASCII

agent.sources.sql-source.type=org.keedio.flume.source.SQLSource
agent.sources.sql-
source.hibernate.connection.url=jdbc:postgresql://<db-ip-
address>/sourcedb?schema=public
agent.sources.sql-source.hibernate.connection.user=postgres
agent.sources.sql-source.hibernate.connection.password=<db-
password>
agent.sources.sql-source.table=customer
agent.sources.sql-source.columns.to.select=*
agent.sources.sql-source.status.file.path=<path-for-status-
file>
agent.sources.sql-source.status.file.name=sql-source.status
```

从上面的配置中可以看到，已经在 Flume 配置文件中配置了两个数据源，即 SQL 源和

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

spool 源。使用者请按真实环境替换下列配置：

- <db-password> 数据库密码。
- <directory containing the spool file> 目录的完整路径，但不应包含 spool 文件名。
- <db-ip-address> PostgreSQL 数据库的 IP 地址。
- <path-for-status-file> 状态文件写入目录的完整路径。

3）下一步是配置 Channel，这里对应的是 Kafka。Kafka 作为一个 Channel，所有数据在摄入 HDFS 之前将以消息的形式流经 Kafka。因为我们有两个 source，所以需要配置两个 Channel，如下所示（在相同的文件中配置，即 customer-data-kafkaChannel-flume-conf.properties）：

```
agent.channels = kafkaCustomerLocationLogChannel
kafkaCustomerDBChannel

agent.channels.kafkaCustomerLocationLogChannel.type
=org.apache.flume.channel.kafka.KafkaChannel
agent.channels.kafkaCustomerLocationLogChannel.kafka.bootstrap.
servers=<kafka-broker-ip>:9092
agent.channels.kafkaCustomerLocationLogChannel.kafka.topic=cust
omerLocation
agent.sources.spool-source.channels =
kafkaCustomerLocationLogChannel
agent.channels.kafkaCustomerLocationLogChannel.parseAsFlumeEven
t = true

agent.channels.kafkaCustomerDBChannel.type
=org.apache.flume.channel.kafka.KafkaChannel
agent.channels.kafkaCustomerDBChannel.kafka.bootstrap.servers=<
kafka-broker-ip>:9092
agent.channels.kafkaCustomerDBChannel.kafka.topic=customer
agent.sources.sql-source.channels = kafkaCustomerDBChannel
```

在上面的配置中，请用真实的运行环境中的 Kafka broker IP 替换 <kafka-broker-ip>，并且为每个数据源定义不同的 topic。

4）现在用以下命令启动 Flume 进程：

```
${FLUME_HOME}/bin/flume-ng agent --conf ${FLUME_HOME}/conf/  -f
${FLUME_HOME}/conf/customer-data-kafkaChannel-flume-conf.properties  -n
agent -Dflume.root.logger=INFO,console
```

如果上述命令成功运行，Flume 进程将通过追加 COMPLETED 的方式对 spool 文件重命名。

5）一旦 Flume 进程启动，可以观察来自数据源的消息，即来自数据库和日志文件中的数据都被发布到各自的 Kafka topic，以供 Flink 进程使用。这可以通过以下命令来验证，这将为 topic 显示相应的队列深度，topic 分别为 customer 和 customerLocation。

```
${KAFKA_HOME}/bin/kafka-run-class.sh kafka.tools.GetOffsetShell --
broker-list <kafka-broker-ip>:9092 --topic customer
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
${KAFKA_HOME}/bin/kafka-run-class.sh kafka.tools.GetOffsetShell --
broker-list <kafka-broker-ip>:9092 --topic customerLocation
```

3. Flink Processor

由于使用了两种不同数据结构的 Source，因此分别将数据发布到两个不同的 topic。这要求在 Flume 中构建两条 Flink 执行流水线，它们是并行工作的。

这里的基础范例代码与 chapter08 / flume-example1 中的代码基本一致，这是因为两个范例中消息都源自 Kafka topic，最后被保存到 HDFS 中。不过两个例子中，消息的数据结构差别很大，因此反序列化模式的实现要做相应的修改，如下所示。代码对应的类可以在 chapter08/flink-customer-db 工程中找到，如下所示。代码结构清晰，便于读者理解。

```
public class Tuple2CustomerProfileMessageDeserializationSchema implements
DeserializationSchema {
private static final ObjectMapper MAPPER = new ObjectMapper();

public Object deserialize(byte[] bytes) throws IOException {
        String message = new String(bytes);
        message=message.replace("\",\"", ",");
        message=message.substring(message.indexOf("\"")+1,
message.lastIndexOf("\""));
        String[] data = message.split(",");
        Customer cust = new Customer();
        cust.setId(Integer.parseInt(data[0]));
        cust.setFirstName(data[1]);
        cust.setLastName(data[2]);
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-
dd");
        LocalDate date = LocalDate.parse(data[3], formatter);
        cust.setDob(java.sql.Date.valueOf(date));
        String customerMessage = MAPPER.writeValueAsString(cust);
        Tuple2<IntWritable, Text> tuple = new Tuple2<IntWritable, Text>();
        tuple.setFields( new IntWritable(cust.getId()), new
Text(customerMessage));
return tuple;
    }

public boolean isEndOfStream(Object o) {
return false;
    }

public TypeInformation<Tuple2<IntWritable, Text>> getProducedType() {
return new TupleTypeInfo<Tuple2<IntWritable,
Text>>(TypeExtractor.createTypeInfo(IntWritable.class),
TypeExtractor.createTypeInfo(Text.class));
    }
}
```

如上面的代码所示，客户信息被反序列化到 Customer 对象中，然后将它序列化为 Tuple2 对象，以便将其写入 HDFS。应传递以下参数来运行 CustomerDBMessageProcessor 类。在命令提示符中输入下列命令以执行范例程序：

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
java -cp target/flink-customer-db-1.0-SNAPSHOT.jar
com.laketravels.ch08.ingestor.customer.CustomerDBMessageProcessor --topic
customer --bootstrap.servers <kafka-server-ip>:9092 --zookeeper.connect
<zookeeper-ip>:2181 --group.id 1 --auto.offset.reset earliest
```

如果一切顺利，此时已经能看到 Flink 正在消费 Kafka topic（customer）中的消息，并将数据持久化到 HDFS 中去，这可以在 Hue 的 HDFS 浏览器中查看运行结果。下面的代码片段显示了针对 Customer Location 的反序列化模式。

```
 public class Tuple2CustomerLocationMessageDeserializationSchema implements
DeserializationSchema {
   private static final ObjectMapper MAPPER = new ObjectMapper();
   public Object deserialize(byte[] bytes) throws IOException {

      //Spooled messages have 2 bytes of leading unicode chars
      String message = new String(bytes, 2, bytes.length-2);
      if (message.trim().length()>0) {
         String[] locationAttributes = message.split(",");
         ObjectNode locationObject = MAPPER.createObjectNode();
         for (String attribute : locationAttributes) {
            String[] attributeElments = attribute.split(":");
            String attributeName = attributeElments[0].replaceAll("""", "");
            String attributeValue = attributeElments[1].replaceAll("""",
               "");
            locationObject.put(attributeName, attributeValue);
          }
          Tuple2<IntWritable, Text> tuple = new Tuple2<IntWritable,
Text>();
          tuple.setFields(new
IntWritable(locationObject.get("id").asInt()),
             new Text(locationObject.toString()));
          return tuple;
       } else {
          return null;
       }
   }

   public boolean isEndOfStream(Object o) {
      return false;
   }

   public TypeInformation<Tuple2<IntWritable, Text>> getProducedType() {
      return new TupleTypeInfo<Tuple2<IntWritable, Text>>
         (TypeExtractor.createTypeInfo(IntWritable.class),
         TypeExtractor.createTypeInfo(Text.class));
   }
}
```

如前面的代码片段所示，客户位置信息的反序列化是非常灵活的，即使未来增加任何额外的属性。

为了处理流经 Flink 的客户位置信息，请传递下列参数给 CustomerLocationMessage Processor 类，代码位于 chapter08/flink-customer-log 工程目录中，在命令提示符中输入以下命令来执行范例程序。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
java -cp target/flink-customer-log-1.0-SNAPSHOT.jar
com.laketravels.ch08.ingestor.location.CustomerLocationMessageProcessor --
topic customerLocation --bootstrap.servers <KAFKA_SERVER_IP>:9092 --
zookeeper.connect <ZOOKEEPER_IP>:2181 --group.id 1 --auto.offset.reset
earliest
```

可登录 Namenode Server 管理页面来查看写入 HDFS 中的数据，可参考图 8-26。登录 http://<hadoop-server-ip>:50070/，浏览 Browse the file system 菜单项。

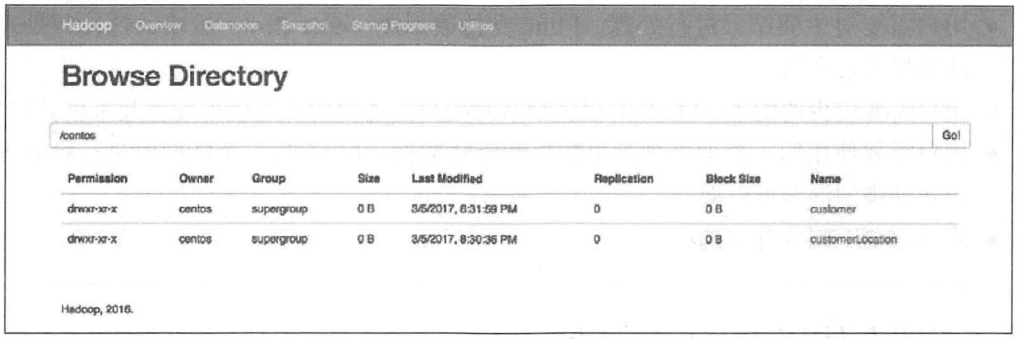


图 8-26　多源数据通过 Flink 存储到 HDFS 中

图 8-27 很好地总结了 SCV 用例是如何与 Flink 技术结合的。在我们的用例中，当数据流入数据湖时，Flink 对流数据进行了适当的处理。

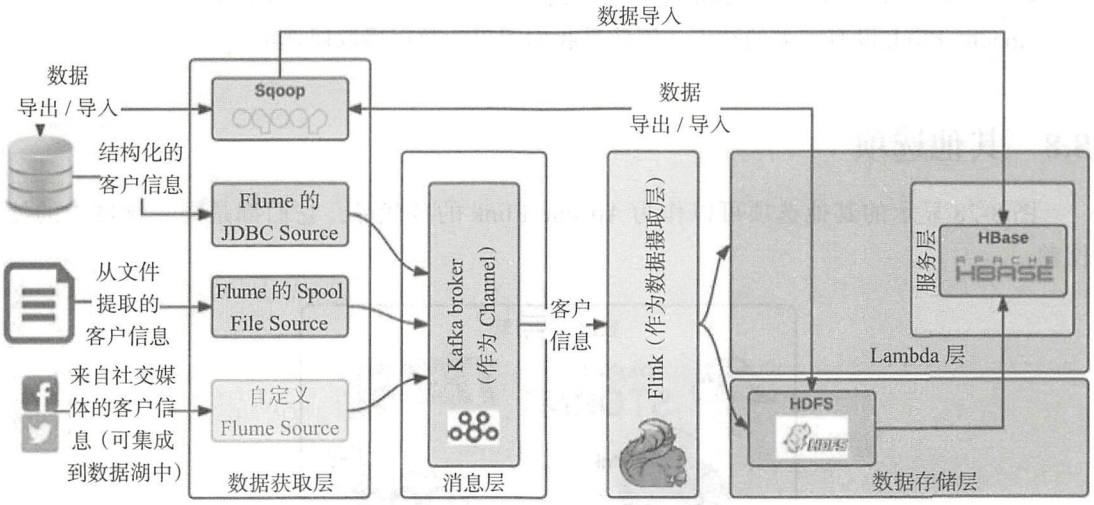


图 8-27　SCV 用例视图中的 Flink

8.6　适合使用 Flink 的场景

可在以下场景中选择 Flink 来处理数据：

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

- 用户需要高性能。目前来看，Flink 是性能最好的流式处理框架之一。
- 用户需要用到机器学习。Flink 的原生闭环迭代操作符能极大地提高处理速度。
- 用户的案例中需要用到图处理。另外，基于前面所述的特性，Flink 能更快地处理数据。
- 用户需要有一致性保障前提下的高吞吐率。
- 用户对每条消息只需要处理一次时。这种处理方式也避免了重复记录处理。
- 用户希望避免手动处理内存，并将该功能交给框架本身。Flink 实现了自动内存管理。
- 用户需要对中间结果进行处理。Flink 支持数据流（data flow）策略，这使得处理中间结果非常容易。
- 用户需要较少的配置。Flink 的很多细节对用户是透明的，这使得配置起来极其简单。
- 用户需要使用同一框架处理批量数据和流式数据。Flink 是一个能同时进行批处理和流式处理的混合型框架。
- 用户需要不同的部署选项。

8.7 不适合使用 Flink 的场景

有以下要求时，避免使用 Flink：

- 如果需要一个更成熟的框架，Flink 与同领域其他框架相比没有优势。
- 如果需要支持 Java 和 Scala 之外编程语言的 API，则不适合使用 Flink。

Apache Flink 没有太多的缺点，因此是我们理想的数据摄取层选项。

8.8 其他选项

图 8-28 显示的其他选项可以作为 Apache Flink 的替代品，它们都是同一领域中的佼佼者。

图 8-28 其他数据处理引擎

8.8.1 Apache Spark

如果不采用 Apache Flink 作为数据摄取层技术框架，那么 Apache Spark 将是最佳选项。

仅供非商业用途或交流学习使用

Apache Spark 是一个开源的集群式计算框架。最开始由加州大学伯克利分校的 AMP 实验室开发，之后 Spark 代码库被捐赠给 Apache 软件基金会，之后一直由该基金会维护。Spark 提供了编程接口用于操作集群，底层默认提供并行数据处理及容错能力。

——维基百科

Apache Spark 是开源社区中最著名的数据处理框架之一，它拥有众多的用户与贡献者。Apache Spark 底层基于微批处理（micro-batching），因此很多人会选择 Apache Flink，因为它能迎合未来的数据处理趋势（很多案例需要实时数据处理能力而不是微批处理能力）。

Spark 可以部署和运行在多种拓扑结构及技术平台上。它还可以集成多个数据源，包括 HDFS、Cassandra 等。通过 Java、Scala、Python、R 等语言调用 Spark API 非常容易，Spark 在这方面比较有优势。此外，它还具有高性能和低延迟的优点。

8.8.2　Apache Storm

Apache Storm 是一个由 Clojure 语言开发的分布式流式处理框架。最初由 BackType 的 Nathan Marz 及其团队设计实现，该项目在被 Twitter 收购以后开源。它使用定制的 spout 和 bolt 来定义信息来源和操作，以允许对流数据进行批量处理。最初的发布时间是 2011 年 9 月 17 日。

——维基百科

Storm 是一个开源的实时数据处理框架，它可以与任何编程语言交互。Storm 与 Spark、Flink 类似，也具有分布式、高性能、容错、可靠消息传递等特点。Storm 中可轻松集成多种技术，这使得它能适合在很多案例中使用。

8.8.3　Apache Tez

Apache Tez 是一种可扩展的框架，用于构建高性能的批处理和交互式数据处理应用程序，通过 Apache Hadoop YARN 进行协调。Tez 通过提升运行速度来显著改进 MapReduce 范式，同时保持了 MapReduce 的可伸缩性（存储能力高达 PB 级）。重要的 Hadoop 生态系统项目，如 Apache Hive 和 Apache Pig 都在使用 Apache Tez，针对 Hadoop 生态系统的第三方数据访问应用程序会越来越多。

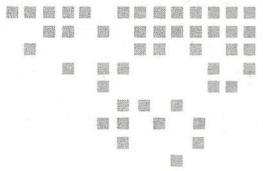
——hortonworks.com（http://hortonworks.com/apache/tez/）

Apache Tez 设计为构建在 Hadoop 2 的 YARN 之上。Tez 是专为高性能、低延迟数据处理而设计的。Tez 开发时基于可扩展的理念，允许用户非常便捷地集成数据传输方面的插件。该框架不断演化的一个重要原因是想消除原生 MapReduce 强加的各种限制。由于 Hadoop 仍然是大数据技术的核心，Tez 在这种情况下是非常有用的，因为它本身就支持 HDFS。

8.9 总结

与本书其他章类似，在本章中，首先介绍技术在数据湖中所处的层级，然后再介绍本层级选用的具体技术，这里采用的是 Apache Flink。我们对 Apache Flink 做了详细的介绍，对 Flink 的架构进行了探究，并简单介绍了该重要框架的核心特性。然后，通过 Apache Flink 技术的实际实现来导出我们的 SCV 用例。最后介绍了何时适合使用 Flink，何时不适合使用 Flink，在本章结尾处，列举了 Apache Flink 的一些竞争对手。

阅读本章后，读者应该已经对数据摄取层及 Apache Flink 的功能有清晰的认识了。同时读者也应该对 Flink 的架构及核心组件的工作机制有非常深入的理解。结合书中的范例，读者也获得了 Flink 实操经验。当然，读者也会对 Flink 的替代品有基本的了解。



第 9 章 Chapter 9

使用 Apache Hadoop 存储数据

数据处理流程无非是这样：先获取数据，然后处理数据，之后再存储数据。本章旨在涵盖数据湖中数据存储相关的重要内容。

构建数据湖时需要遵循的一个核心原则是，所有数据以最原始的形式存储在数据湖中，而不是仅仅存储处理过或清洗过的数据。这个原则非常关键，因为有些数据在当时看起来不重要，在后期可能变得非常重要，届时可以利用这些存储好的原始数据。

本章和其他章节类似，将首先介绍数据存储层，然后进入技术路线图部分。之后，我们将深入研究被选用的技术框架，确保读者对该技术的诸方面有深入的了解。

正如本章的标题所示，这里选择的技术框架为 Apache Hadoop，用于在数据湖中存储原始格式的非索引数据。与其他章节一样，本章将先介绍在数据存储层选择该技术的理由，然后深入讨论其架构。之后将介绍 Hadoop 框架中的各种组件和工具。此后，本章将通过实际的代码带领读者着手构建数据存储层。最后，本章将介绍一些读者可以用来代替 Hadoop 的其他选项，并且会给出 SCV 用例实现的最终架构。

请准备好，我们将开始尝试存储获取的各种数据。

9.1 数据湖背景中的数据存储和 Lambda 批处理层

在数据湖的实现中，我们设立了一个专门的层次，用于持久化存储数据，这就是数据存储层。数据从各种数据源收集起来，持久化存储在各种数据存储系统中，这些存储系统能够处理不同类型和格式的数据。在本章中，将介绍如何将非索引格式的原始数据存储在数据湖中。

这里选择 Apache Hadoop 作为数据存储的技术选项。我们认为，在该层选择这项技术不会有太多争议，因为它具有诸多奇妙的特性。此外，该技术在问世不久之后就表现出了惊人的成熟度和吸引社区、企业支持的能力。

本章中随后的部分旨在详细介绍 Hadoop，以便读者能清楚地了解该技术并深入了解数据存储层。

9.1.1 数据存储和 Lambda 批处理层

在第 2 章中，读者已经对数据存储层有了一定的认识。该层的职责是将收集的数据保存到数据湖的永久性存储中。Lambda 架构的批处理层的职责是为存储在数据存储层中的数据创建批处理视图。图 9-1 将加深读者的记忆，为读者提供该层的图形化表示。

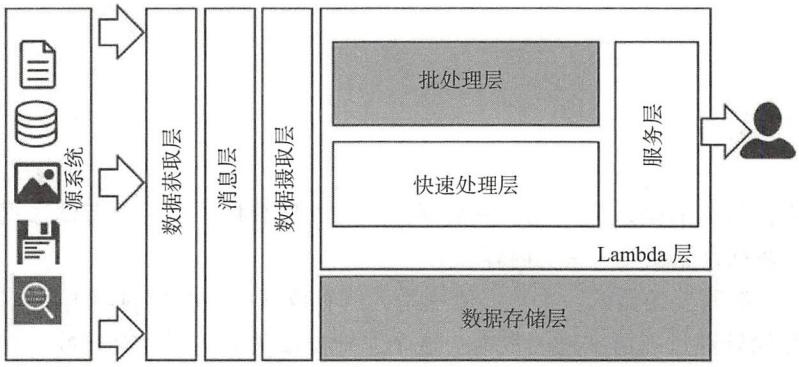


图 9-1 数据湖——数据存储层和 Lambda 批处理层

数据存储层是一个非常重要的层，它对来自不同数据源系统的不同类型的原始数据进行持久化存储，同时能根据需求轻松进行扩展。对于该层来说，把 IOPS（参见下面的条目解释）定义清楚非常重要，因为它是数据源系统中的数据进入数据湖的数据量和速度的一个决定性因素。事实上，Hadoop 正是致力于存储非结构化数据的技术之一。还有其他的机制，如 NoSQL（参见下面条目解释）和 NewSQL（参见下面的条目解释）。

每秒输入/输出操作（IOPS）是用于表征计算机存储设备（如硬盘驱动器（HDD）、固态驱动器（SSD）和存储区域网络（SAN））性能的指标。

NoSQL（最初是指 non-SQL，non relational 或 not only SQL）数据库提供了一种数据存储和检索机制，该机制与关系型数据库中基于表格关系进行建模的方式有所不同。

NewSQL 是一类现代关系型数据库管理系统，试图为联机事务处理（OLTP）读写工作模式提供与 NoSQL 系统相同的可伸缩能力，同时仍保持传统数据库系统的 ACID 特性。

——维基百科

存储层应该能够处理以下内容：

- 支持各种查询分析工具。
- 不同模式下的多种类型的数据（批量和实时）。
- 不同格式的数据，例如结构化数据、非结构化数据和半结构化数据。
- 不同规模的需求。
- 采用各种数据压缩技术保证持久化的效率。
- 不同的数据处理速度（KB/s，MB/s 等）。
- 视情况而定，提供不同的查询机制和语言支持，使得用户能从数据湖中提取相关数据进行各种分析。

9.1.2　数据存储和 Lambda 批处理层——技术路线图

为了涵盖本书的用例并构建数据湖，使用了多种存储系统。本章旨在讨论非索引格式原始数据的存储机制。本章将深入研究 Hadoop，这是我们处理非索引数据存储的技术选项。

图 9-2 中介绍了本书遵循的概念架构，以及其中技术相关的各个方面。在第三部分的内容将所有技术整合起来之前，本书将继续介绍每种技术及其在整体架构中的作用，以及它们之间的关联。

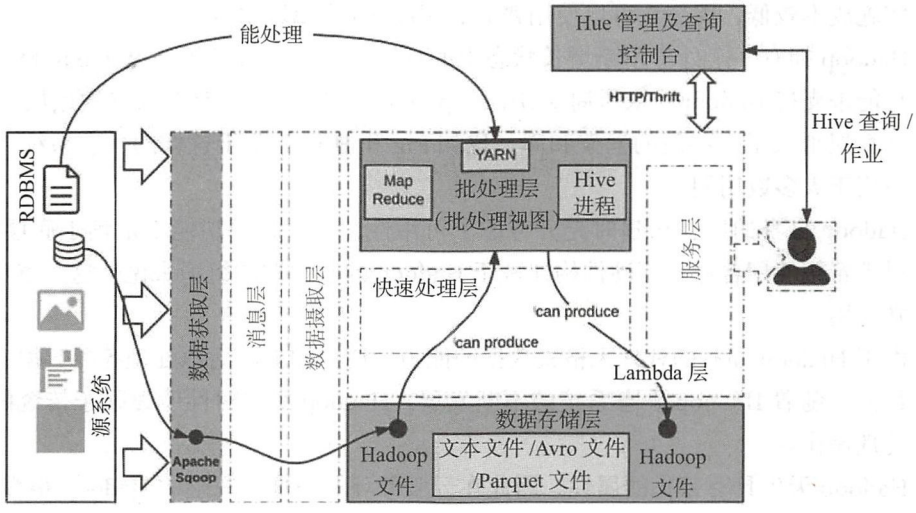


图 9-2　数据存储和 Lambda 批处理层技术路线图

对 SCV 用例而言，已经从各种数据源系统中收集数据，并且将这些原始数据保存在 Hadoop 中，如图 9-2 中所示。

本章的后续小节将提供更多与 Hadoop 相关的细节内容，以便让读者了解 Hadoop 架构

中的各个重要组件。

9.1.3 什么是 Apache Hadoop

在详细介绍 Hadoop 之前，希望读者对 Hadoop 有一个大概的认识。这是本书介绍各种技术时一直遵循的方法。

Apache Hadoop 是一种计算框架，在该框架中计算机集群（并非必须是服务器，也可以是日常生活中使用的普通商用计算机）可以进行分布式计算，同时也可以存储大量不同格式的数据。读者也不难推测，它也可以用于数据计算，这是 Hadoop 的一类基本功能。我们会用到 Hadoop 的计算能力，但是更重要的是利用它的分布式存储数据的能力。

由于其强大的功能和极高的普及度，几乎所有组织都将 Hadoop 纳入了它们的技术栈中。本书确信读者所在组织已经掌握了这一框架，因此可以轻易应用到数据湖建设中。

9.2 为什么使用 Hadoop

在笔者看来，使用 Hadoop 的原因是显而易见的。就行业现状而言，在大数据领域中 Apache Hadoop 是不可或缺的。的确还有其他选项，但其中大数据都需要与 Hadoop 协同工作。在数据湖的实现中，我们把 Hadoop 作为首选的数据存储技术，主要原因如下：

- Hadoop 可以轻松地处理大量的结构化、半结构化和非结构化数据。
- 实施成本较低，因为它可以使用普通商用硬件并根据需求伸缩。
- Hadoop 拥有一个处于不断增长状态中的社区，该社区通过频繁提交 bug 修复和增强功能来支持 Hadoop。众所周知，Hadoop 有两个核心层，即计算层和数据层（HDFS）。计算层加入了一些新的框架和库，比如 Pig 和 Hive，基于这样的生态系统，Hadoop 适用于大多数用例。
- Hadoop 自身的库在构建时充分考虑了可用性，并且它的功能不依赖于底层的硬件。对于希望使用普通商用硬件构建基于 Hadoop 的数据湖的组织来说，这一点让他们非常受用。
- 由于 Hadoop 固有的处理无格式数据的能力，使得它能灵活地处理各种数据。
- 目前，随着 Hadoop 生态系统的不断发展，Hadoop 的实时性明显强于传统的数据批处理操作。
- Hadoop 天生具有云化的属性，因此在云服务环境中部署完整的 Hadoop 很容易实现，并且对于在向大数据和数据湖方向小跑前进的组织来说，这样操作在成本收益考量上更为经济。
- Hadoop 天然具有健壮性和容错能力。
- Hadoop 拥有强大的计算能力，因此非常适合用于计算密集型的分析需求。
- Hadoop 具有很高的运算速度，可以很容易地处理复杂的计算逻辑。

- Hadoop 可以运行在大多数操作系统上，例如 Linux、Mac、Windows 和 Solaris。

9.2.1　Hadoop 简史

Doug Cutting 从 Google 的 MapReduce（发表于 2004 年，其中应用程序被分解为多个块，然后在同一个集群的多个节点上运行）获得灵感后，创造了这一框架并将其命名为 Hadoop（2006 年），这个名字来源于他孩子的黄色大象玩具。他认为，这个名字很短，又很独特，拼写和发音也相对容易。同年（2006 年），Doug 加入雅虎，并在 2008 年构建了一个由 4000 个节点组成的 Hadoop 集群，雅虎使用该集群打破了 TeraByte 排序基准记录（http://sortbenchmark.org/）。图 9-3 以更加直观的方式展示了 Hadoop 的发展史。

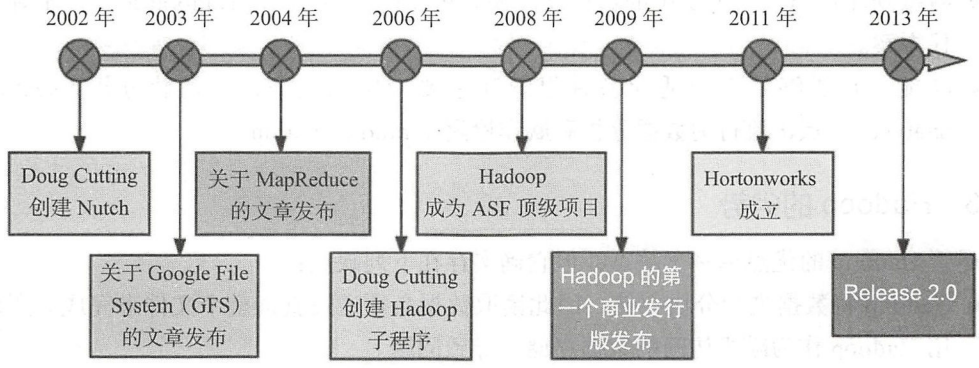


图 9-3　Apache Hadoop 简史

下文将更详细地对图 9-3 进行解读。Doug Cutting（Hadoop 的作者）在 2002 年创建了 Nutch，Nutch 是一个网页采集和搜索系统。在 2003 年，Sanjay Ghemawat、Howard Gobioff 和 Shun Tak Leung 发表了关于 Google 文件系统（GFS）的文章。此后，Sanjay Ghemawat 和 Jeffrey Dean 在 2004 年又发表了另一篇文章（《MapReduce: Simplified Data Processing on Large Cluster》）。受到 MapReduce 论文的启发，Doug 重构了 Nutch 项目，并在 2006 年创建了 Hadoop 子项目。Owen O'Malley 是 2006 年第一位加入 Hadoop 项目的贡献者。Hadoop 的 0.1.0 版本于 2006 年 4 月发布。2008 年，Hadoop 成为 Apache Software Foundation（ASF）的顶级项目，并于 2009 年发布了 Hadoop 的第一个商业发行版。2011 年，Rob Bearden 和雅虎合作成立了 Hortonworks，包括了 Hadoop 团队的主要成员，有 Arun Murthy、Devaraj Das、Sanjay Radia、Suresh Srinivas、Alan Gates 以及最重要的 Owen O'Malley。

9.2.2　Hadoop 的优势

前文中涵盖了选择 Hadoop 的部分原因，这些也正是 Hadoop 的核心优势。下文将以更清晰易读并且更为细致和深入的方式重新表述这些因素。

- 可伸缩性：能够处理多种类型的海量数据。用户可以添加更多节点来处理更多数据

（可线性伸缩）。

- 灵活性：数据没有结构化模式（无模式）。它能够处理不同类型的数据（大数据 4V 之一的 variety）。
- 高性能：能够高速处理海量数据。
- 低成本：能够在普通商用硬件上运行，使得其实施成本较低。Hadoop 是开源软件，并且拥有一个充满活力的社区，它处于不断的发展变化中而不会停滞不前。
- 高级分析：能够为高级分析的需求进行计算并输出。
- 庞大的生态系统：Hadoop 生态系统，尤其是其计算层，随着新的开源项目不断推出而迅速发展。
- 数据仓库：当一个组织中的数据仓库需要极大的灵活性时，Hadoop 是一个很好的替代方案。
- 功能：它能够处理或适配各种组织和重要用例的业务，如情感分析（sentiment analysis），点击流行为数据分析和欺诈检测（fraud detection）。

9.2.3 Hadoop 的劣势

尽管 Hadoop 的优点远多于缺点，但它确实存在下列缺点：

- Hadoop 将数据进行分块存储，因此读取数据必须通过查询整个文件来完成。所以使用 Hadoop 作为随机访问的数据存储会导致问题。
- 如果用户的数据量较小（本书的用例中不是），那么使用 Hadoop 可能会很麻烦，或者无法获得预期的好处。
- 一些高级算法依赖更具体的硬件需求，用 Hadoop 执行这些算法可能会出现问题。
- 在早期，相关的技能熟练的人才较少，但是市场上这种技能日益增加。然而，这仍然是组织中 Hadoop 生态系统的一个缺陷。
- 存在安全隐患，因为 Hadoop 没有考虑到企业级安全方面。所以，这种问题往往出现在将 Hadoop 用于存储企业级客户信息时（如 PII，条目解释参考下文）。
- 工具的缺乏得到了改善，但这仍然是一个问题，因为许多方面还需要手动编码实现（技能在这里起作用）。
- Hadoop 可以运行在通用硬件上，但为了构建企业级系统，该组织仍然会投入昂贵的硬件。因此，尽管理论上它的成本较低，但对企业来说，成本依然高昂。

个人身份信息（Personally Identifiable Information，PII）或敏感个人信息（Sensitive Personal Information，SPI），应用于信息安全和隐私法律中，指的是可以单独使用或者与其他信息一起使用的，用以识别、联系、查找某个人或者在上下文中确定某个人的信息。

——维基百科

9.3　Hadoop 的工作原理

本节中将介绍 Hadoop 及其组件、架构以及工作原理。我们首先了解一些 Hadoop 的核心架构原则，然后再详细介绍它的架构和一些重要组件。

9.3.1　Hadoop 架构的核心原则

Hadoop 的构建和构思具有明确的架构目标和原则，具体如下（以下内容不具有权威性，因为无法找到确切的出处，以下内容从 https://goo.gl/3nvERl 收集而来）：

- 线性可伸缩（水平扩展而不是垂直扩展）：添加更多节点以增加数据存储和计算能力。
- 传输代码而不传输数据：在大数据场景中，数据量通常很庞大，数据处理的代码很少。因此，本原则规定，将代码引入或分发到需要处理数据的节点 / 机器上，而不分发或移动数据。实质上，它意味着最大限度地减少数据传输而代之以代码分发。
- 失败很常见并需要处理：通过实际预防和处理这些情况来提高可靠性和容错能力。
- 简单的计算模型：可靠性和容错依赖分布式和并发能力；对用户隐藏这些细节并提供一个抽象的层来处理。这是 Hadoop 被广泛采用的主要原因之一。
- 数据顺序处理：避免数据随机访问读取。
- 自动化管理：自动管理分布式应用程序 / 框架的多个通用方面，而不依赖于人工干预。
- 并行处理：默认情况下，以自动方式进行并行处理。

水平扩展指的是添加更多的节点 / 机器，但这些机器内存较少、处理性能较差。从成本角度考虑，这是一个更实际的选择。

垂直扩展是指向现有节点 / 服务器添加更多内存和处理能力。一般来说，与水平扩展相比，该方案成本更高。在一段时间内，这是一个不太实际的选择，还可能导致维护问题。

图 9-4 展示了水平扩展与垂直扩展的架构。

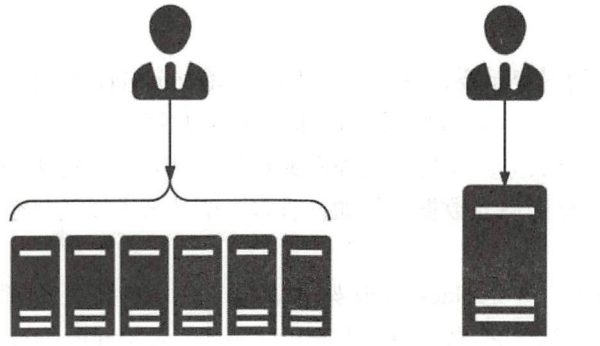


图 9-4　水平扩展与垂直扩展架构

9.3.2 Hadoop 架构

下面开始介绍 Hadoop 的架构。本书涵盖了 Hadoop 1.x 和 Hadoop 2.x 两个版本，来让读者可以身临其境地去体会这个美妙的框架的演变过程。在本书的用例实现和示例中，使用的是 Hadoop 2.x 版本。

1. Hadoop 1.x 版本的架构

了解第一代的 Hadoop 很有必要，因为许多方面都和 MapReduce 组件捆绑，计算执行也依赖 MapReduce，比如资源管理、作业调度和作业处理。为了解决这个问题，第二代 Hadoop 出现并且在被广泛使用。想要了解 Hadoop 2.x 的架构，最好先去了解 Hadoop 1.x 架构的基础知识，这就是本节将简要介绍的内容。

图 9-5 显示了 Hadoop 1.x 框架的概念架构。

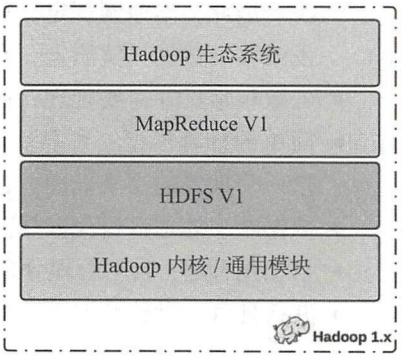


图 9-5 Hadooop 1.x 高层次概念架构

我们将自下而上地深入研究图 9-5 中的每个组件：

- Hadoop Core/Common Module：这一层包含了所有图 9-5 中上层组件使用的基础 Hadoop API。由于 Hadoop 是用 Java 编写的，这一层的功能与 Hadoop 一起打包到一个 JAR 文件中（参见下面的条目解释）。这是其他组件所需的必需组件，它包含可重用的代码和工具。
- HDFS（V1）：它是 Hadoop 文件系统（Hadoop Distributed File System）的简称。需要注意的是，在 Hadoop 1.x 中，正在使用的是 HDFS 的 V1 版本。它也被称为 HDFS V1，我们将在本书中继续使用这个名称。这是负责为 Hadoop 提供分布式存储功能的核心组件。其默认块的大小是 64MB，也可以根据用户的用例情况灵活更改。HDFS V1 分为两个子组件：
  - NameNode
  - DataNode

知道它的实际部署情况很重要，它确实有主从（Master-Slave）概念。在 HDFS V1 中，NameNode 存在于主节点中，负责存储用于 Hadoop 的元数据。

DataNode 驻留在从节点上，并将应用程序的数据存储在定义好大小的数据块中（默认值为 64 MB）。NameNode 存储元数据，例如，有多少个从节点以及每个数据节点中数据块的数量。

- MapReduce（V1）：这是 Hadoop 框架中的分布式数据处理系统。基于 Google 的 MapReduce 算法，它也被称为 MRV1 或经典 MapReduce。与 HDFS 类似，MapReduce 也有两个子组件，也遵循主从模型，如下所示：

仅供非商业用途或交流学习使用

- JobTracker
- TaskTracker

与 HDFS 组件非常相似，JobTracker 驻留在主节点而 TaskTracker 驻留在从节点中。顾名思义，JobTracker 将任务分配给 TaskTracker，并记录和维护每个 TaskTracker 的状态。另一方面，TaskTracker 执行分配的任务并在执行后将状态发送回 JobTracker。

- HadoopEcosystem：Hadoop 生态系统，各种 Hadoop 外部工具和库都基于核心组件工作，如 HDFS V1 和 MapReduce V1。

JAR 文件是 Java 运行时环境（JRE）使用的 Java 归档文件，JRE 是用于执行 Java 程序的框架。JAR 文件可用作程序库或作为独立程序运行，只要 JRE 安装在计算机或移动设备上。

——https://fileinfo.com

图 9-6 揭示了更详细的 Hadoop 1.x 概念架构，它是对前面高层次架构的更细致的阐述。

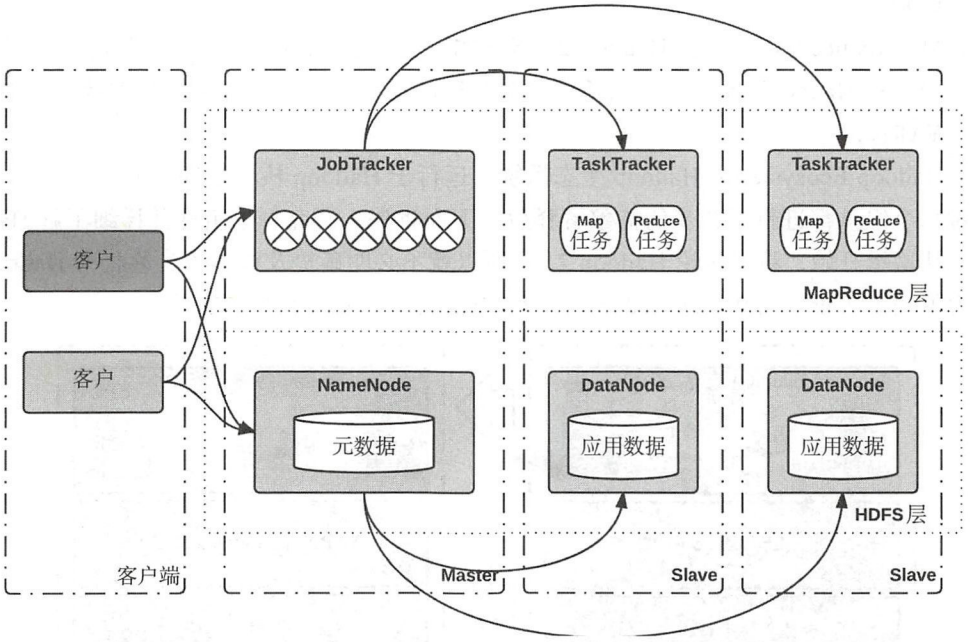


图 9-6　Hadooop 1.x 低层次概念架构

图 9-6 浅显易懂。读者已经知道，主节点和从节点都有两个核心组件：HDFS V1 和 MapReduce V1。在主节点和从节点上，这些核心组件中的各种子组件变得活跃，并支撑 Hadoop 的完整功能运转。从节点的 HDFS 子组件 TaskTracker 包含两个任务，即 Map 任务和 Reduce 任务，与 MapReduce 算法内联。

仅供非商业用途或交流学习使用

2. Hadoop 2.x 版本的架构

到现在为止，希望读者已经清楚理解了 Hadoop 1.x 的架构，现在让我们去探究本书案例中使用到的 Hadoop 2.7.3（2.x）版本吧。

图 9-7 抽象地显示了 Hadoop 2.x 架构的各个组件。

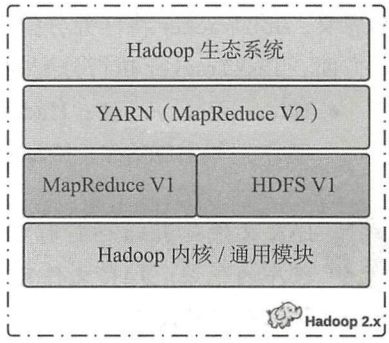


图 9-7 Hadoop 2.x 的高层次概念架构

Hadoop 2.x 架构的核心方面与 Hadoop 1.x 非常相似。如图 9-7 所示，Hadoop 架构中包含如下主要组件：

- Hadoop Core/Common Module：如前文所述，它指的是 JAR 文件形式的核心模块，所有其他模块都依赖它来完成其核心功能。
- HDFS（V2）：在 HDFS V1 的基础上增加一些高级特性构成了 HDFS V2。
- YARN（MR V2）：YARN（Yet Another Resource Negotiator）成为一个核心组件，这也是 Hadoop 2.x 与 Hadoop 1.x 的一个区别。
- MapReduce（V1）：在 Hadoop 2.x 架构中也使用了 Hadoop 1.x 中相同的组件。然而，在 Hadoop 2.x 中，它只负责数据处理的工作，其余所有工作都被转交给强大的 YARN。
- Hadoop Ecosystem：Hadoop 生态系统，运行于 Hadoop 核心组件之上。

对每个组件都进行了初步的抽象解释后，下面将深入研究每个组件并详细了解 Hadoop 架构。从这里开始，将只讨论 Hadoop 2.x，所以就不必明确提及版本了，称之为 Hadoop。

为了便于比较，图 9-8 将 Hadoop 1.x 和 Hadoop 2.x 架构组件并排展示。

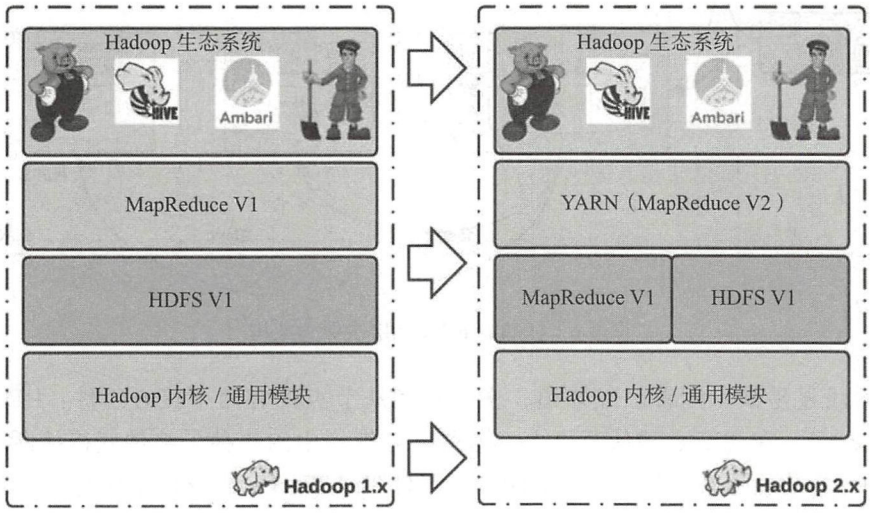


图 9-8 Hadoop 1.x 与 Hadoop 2.x 架构对比

仅供非商业用途或交流学习使用

现在本书开始详细介绍 Hadoop 2.x 的各个核心组件。最后将详细解释 Hadoop 2.x 架构，类似于前文中对 Hadoop 1.x 的架构的介绍。

9.3.3　Hadoop 架构组件

本节将深入介绍 Hadoop 的各个组件。

1. HDFS

如前所述，Hadoop 的数据存储和计算都遵循主从架构模式。HDFS 作为数据存储的主要组件，有两个子组件，即 NameNode 和 DataNode，分别驻留于主节点和从节点中。DataNode 存储应用程序数据，NameNode 存储文件系统元数据。NameNode 和 DataNode 之间的通信是基于 TCP 进行的，而且可靠性和性能都相当高。

图 9-9 中显示了 HDFS 中使用的主从架构，其中包含 NameNode 和 DataNode 组件。

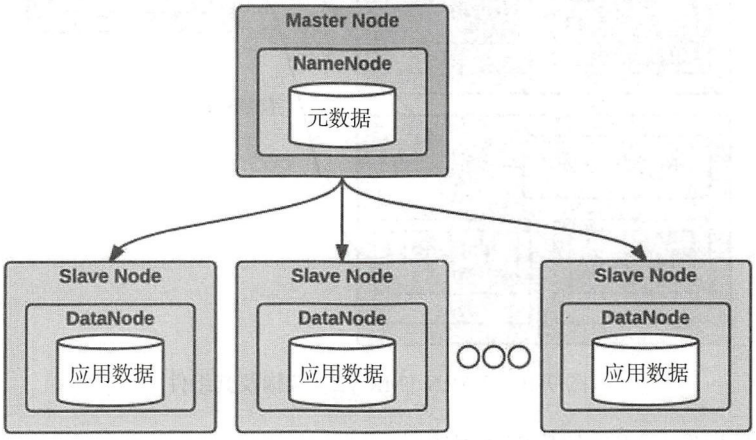


图 9-9　HDFS 的主从架构（这里展示了 NameNode 与 DataNode）

HDFS 是类似于 Google 文件系统（GFS）的分布式文件系统。应用程序数据被分成多个固定大小的数据块（默认情况下为 64 MB，但可以根据需要进行配置），然后分发到 Hadoop 集群中的多个节点，根据集群中的复制因子（replication factor）设置进行对应的数据复制。这种副本机制使得 Hadoop 具有容错性和可靠性。

HDFS 文件系统允许用户按顺序写入一次，并且禁止随机写入。此外，HDFS 本身并不适合随机访问数据，但对于流式读取文件进行了高度优化。

2. YARN

Hadoop（与 Hadoop 1.x 相比）的一个方便之处是使用了资源协调器 YARN，使 Hadoop 能够以动态的方式利用资源，使得应用程序可以完成工作，而不是试图去计算任务对资源的影响。它允许 Hadoop 有效和高效地利用集群中的每个节点。Hadoop 2.x 中的 YARN 也称为 MR V2（MapReduce V2）。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

图 9-10 详细显示了 YARN 的架构及其组件以及它们的工作原理。

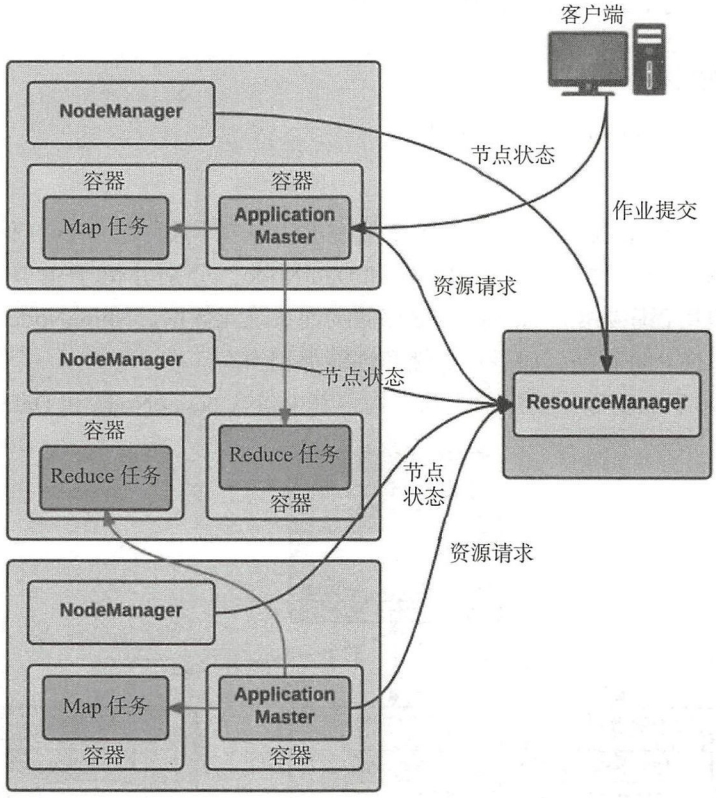


图 9-10 YARN（MR V2）架构及其组件

以下部分是构成 YARN 架构的重要组件：

- ResourceManager（RM）：它是集群节点内管理资源的主要代理，对资源进行相应的分配。它有效地管理集群资源，同时也是 YARN 的主节点和真正的协调器。ResourceManager 有两个子组件，分别是：
  - 应用管理器（application manager）。
  - 调度器（scheduler），响应应用程序的调度需求，同时监控并跟踪。
- NodeManager（NM）：每个节点都有一个 NodeManager，负责节点内的资源分配和处理。该组件也被认为是 YARN 中的从节点。它接受来自资源管理器的请求并报告节点的运行状况和资源。
- Application Master（AM）：每个应用程序有对应的 Application Master，并被委托管理应用程序生命周期，负责任务的分配和执行。

3. MapReduce

Hadoop 框架能够处理大型数据集的一个重要原因就是使用 MapReduce 编程模型。同

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

样，MapReduce 的执行也依赖主从概念，其中输入文件首先被拆解为更小的文件，然后每个文件被传送到工作节点，这些工作节点处理（Map 任务）数据，然后主服务器收集数据（Reduce 任务）并将其发回。图 9-11 描述了上述工作流程。

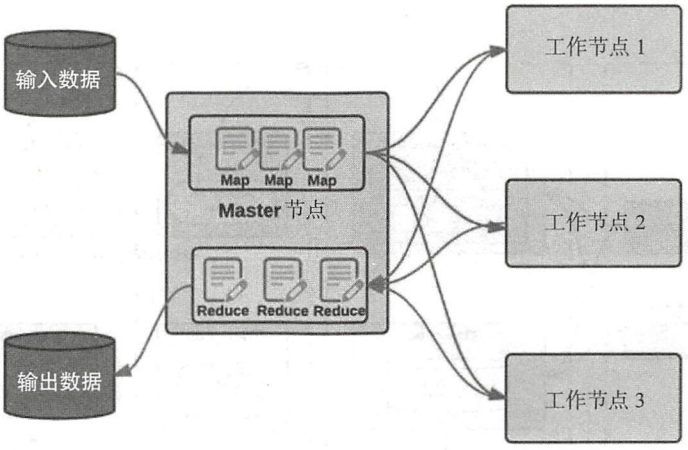


图 9-11　Hadoop MapReduce 编程模型的执行

如图 9-11 所示，Map 将查询（代码向数据迁移）发送到节点，然后收集结果，整理后发回请求端。在这里，YARN 负责并行处理，MapReduce 提供了一个框架，基于该框架将代码（查询）分发到多个节点上执行 / 处理。MapReduce 是一个基于 Java 的编程模型，受到 Google 启发而问世。

4. Hadoop 生态系统

值得专门用一个小节来介绍 Hadoop 生态系统本身，因为在这个主题下涵盖了很多内容。下面的内容就涉及了该主题。

9.3.4　Hadoop 架构细节

前一节中已经详细介绍了 Hadoop 的各个组件。接下来，将详细探讨 Hadoop 2.x 的架构，了解它的各个组件及子组件。

图 9-12 剖析了 Hadoop 2.x 架构的组件和子组件。

Hadoop 1.x 框架的主要问题是存在单点故障隐患，也就是 Hadoop 1.x 中的 NameNode，Hadoop 2.x 成功地解决了这一问题。

这主要是通过将 YARN 框架作为一个组件包含进 Hadoop 2.x 架构中来解决的。从前面的架构图可以看出，ResourceManager 负责管理应用程序，同时还提供了作业管理调度的功能。ResourceManager 与 NodeManager（每台机器的代理）协调工作。NodeManager 负责管理机器的容器、资源监控和报告。ResourceManager 使用这些信息来给各个应用程序分配资源。ApplicationMaster 针对每一个应用程序启动一个进程，该进程负责与 ResourceManager

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

协调资源并与 NodeManager 协同工作来执行和监控任务。调度器负责根据资源需求为各个应用程序分配资源。调度器也跟踪状态或监控应用程序。ApplicationManager 与 ApplicationMaster 协作管理作业提交和故障恢复。

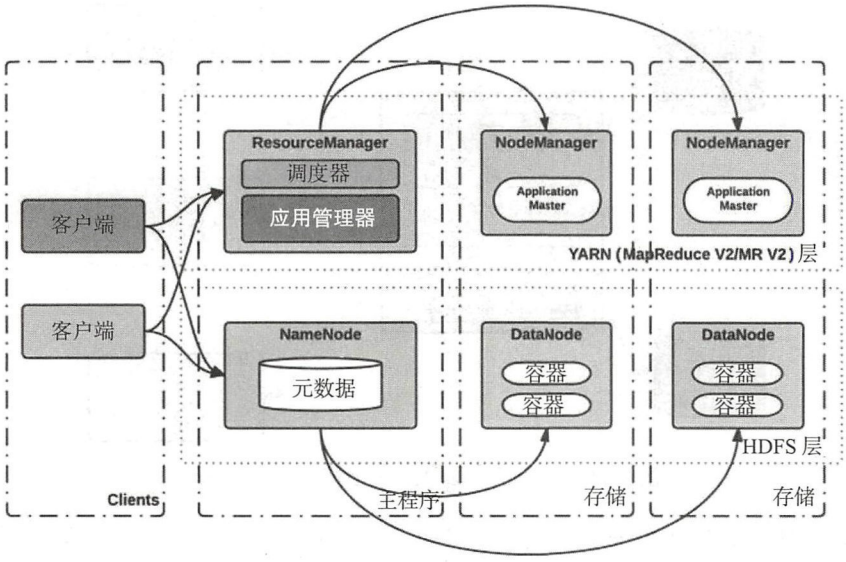


图 9-12 Hadoop 2.x 低层次概念架构

基于这种架构，Hadoop 2.x 中的 YARN 框架消除了作业管理对 NameNode 的重度依赖，从而完全克服了单点故障。

9.4 Hadoop 生态系统

除了核心组件外，Hadoop 还囊括了许多构建在核心组件之上的工具和库，统称为 Hadoop 生态系统。

图 9-13 仅列出了 Hadoop 生态系统中的一些常用框架。

下面将尝试根据这些基于 Hadoop 的框架的核心功能进行分类，并进行简要介绍。对这些框架的所有细节都深入了解是不可能的，这部分内容旨在对相关知识进行大致介绍，以便读者能了解 Hadoop 生态系统的功能和选择。

以下部分涵盖了 Hadoop 生态系统中的各种框架，并将它们按照功能分类。这些功能类别如下：

- 数据访问 / 处理组件
- 数据存储组件
- 监控、管理和协调组件
- 数据集成组件

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

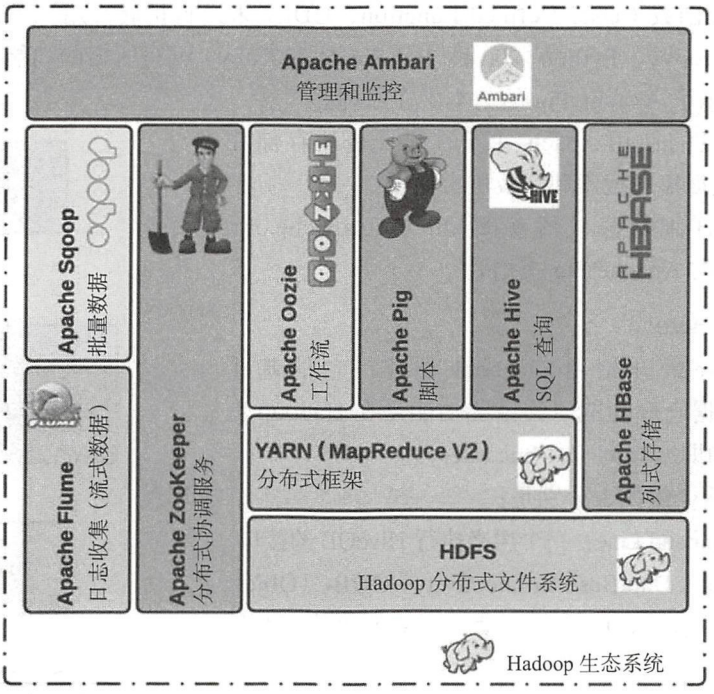


图 9-13　Hadoop 生态系统（只展示一些非常重要的组件）

除了本书讨论的组件外，还有许多由开源社区创建和维护的组件，但本书倾向于讨论更为突出和广为人知的组件。

9.4.1　数据访问 / 处理组件

Hadoop 生态系统中有部分组件，用户可以用它们来访问和处理 HDFS 中存储的数据，本节介绍的组件属于此类。我们将讨论其中一些比较知名的组件，来了解它们的工作原理。

1. Apache Pig

Apache Pig 是由雅虎开发的用于数据访问和处理的平台，该平台基于 HDFS 来处理大型数据集。Pig 有两个组件，分别是：

- 一种被称为 Pig Latin 的高级数据流语言，它具有类似于 SQL 的命令结构。
- Pig 运行时，执行 Pig Latin 语言的运行环境。

Pig 作业对 MapReduce 的复杂性做了抽象，它在后台提交 MapReduce 作业并按顺序执行。除了 MapReduce，Pig 的 Hadoop 作业还可以使用 Apache Tez 和 Apache Spark 来执行。Pig 为 Hadoop 生态系统提供了一种数据流功能，可以将 ETL 功能抽象出来。Pig 允许用户从 HDFS 中提取大型数据集，然后它可以进行某些必要的操作（例如分组和过滤），再根据用例的需求将计算结果写回到 HDFS 或者将结果存储起来。Pig 的数据流语言通过允许用户

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

编写用户自定义函数（User Defined Function，UDF）来进行扩展，用户可以使用该函数以各种语言（如 Groovy、Python、Java、JavaScript 和 Ruby）编写所需的功能。

图 9-14 显示了 Apache Pig 的基本工作原理。

一个更常见的使用 Apache Pig 的例子是从存储在 HDFS 的大型数据集中搜索和屏蔽机密数据。

有关更多详细信息，请参阅 http://pig.apache.org/docs/r0.16.0/ 上的 Apache Pig 文档。

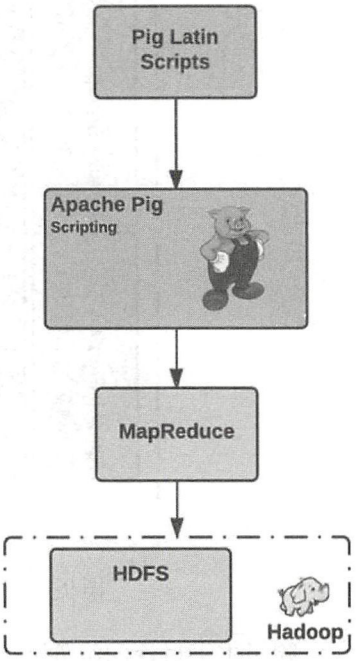


图 9-14 Apache Pig 工作原理

2. Apache Hive

Apache Hive 最开始由 Facebook 创建，它能基于 Hadoop 提供数据仓库功能。其主要功能是在 Hadoop 上执行数据汇总和即席查询（ad-hoc query）。

Hive 包含两个组件，分别是：

- Hive Command Line：一个用来执行 HiveQL 的接口。
- JDBC（Java DataBase Connectivity）/ODBC（Object DataBase Connectivity）驱动：用来创建数据存储的连接。

执行查询是通过 Hive 查询语言（HQL 或 HiveQL）完成的，它和 SQL 非常相似。查询性能很高并且是实时的，而且能使用各种索引功能。Apache Hive 能够进行批处理和实时数据处理。

与 Apache Pig 类似，Hive 也允许用户编写用户自定义函数（User Defined Function，UDF），用户自定义聚合函数（User Defined Aggregate Functions，UDAF）和用户自定义表函数（User Defined Table Functions，UDTF）以满足特定的用例需求。同样，与 Pig 类似，执行查询可以按照用户的要求选择使用 MapReduce、Apache Spark 或者 Apache Tez。

Apache Hive 不适合进行在线事务处理（OLTP），相反，它更适合数据仓库功能（OLAP——在线分析处理）。但是，它能够很容易地处理 PB 级的海量数据。

Hive 适用的主要用例是即席数据分析和报表处理。Hive 对一些知名 BI 工具提供了很好的支持，比如 MicroSTRategy（MSTR），Tableau 和 BO（Business Object）。

更多相关信息，请参阅 http://hive.apache.org/ 上的 Apache Hive 文档。

9.4.2 数据存储组件

Hadoop 生态系统中的一些组件，既可以存储数据，又可以执行抽象查询，本节讨论的组件属于这个类别。接下来探讨一些非常知名的组件，并介绍它们的工作原理。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

Apache Hbase

Apache HBase 依赖 Hadoop，使用 HDFS 作为底层的数据存储组件。HBase 也是非关系型的（NoSQL）、天然分布式的、面向列族（column family）的数据库。它很适合随机读取和批量处理。HBase 能够处理百万级行和列的大型数据集。

Apache HBase 借鉴了 Google 的 Bigtable，它被业界认为是最佳的实现方案之一，其内部实现其实是一个有序的映射结构。

HBase 有多种 API，主要是 Java API。此外，它还有 REST 接口（用于 HTTP 访问）和 Thrift（用于其他语言编程访问）的 API。

HBase 非常适合处理实时数据分析的用例，此外，它也非常适合用于实时数据监控。在本书的例子中，我们使用的是 HBase 的 1.1.8 版本。

有关更多详细信息，请参阅 http://hbase.apache.org/ 上的 Apache HBase 文档。

9.4.3　监控、管理和协调组件

Hadoop 生态系统中能够监控、管理和协调其他部件的组件属于本类。接下来将讨论一些非常知名的组件，并了解它们的工作原理。

1. Apache ZooKeeper

Apache ZooKeeper 是一个流行的、开源的、分布式的协调服务。ZooKeeper 是一种集中式的服务，能够执行多种操作，例如配置管理和其他一些分布式系统常见的协调服务。

ZooKeeper 部署在一组服务器（理想情况下为奇数台）上，这组机器通常称为集群。当集群启动时，其中一台服务器被选为主节点（Leader），其他服务器自动成为从节点（Follower）。驻留在每台服务器中的数据（称为状态）被广播，因此每个服务器都持有它最新的状态。完成主节点选取后，所有的写请求都会发送给主节点，所有的从节点都会从主节点服务器中获取数据。不管出于任何原因导致主节点宕机，集群会选出新的主节点并继续运行。客户端只连接到集群中的一台服务器。客户端与 ZooKeeper 服务建立一个连接，所有请求都会通过该会话基于 TCP 来发送。ZooKeeper 会话处理传入的请求并让它们遵循 FIFO（先进先出）模式（队列）。图 9-15 显示了 ZooKeeper 基本的工作原理。

许多知名的项目使用了 ZooKeeper 来实现分布式协调服务功能。其中一些如下：

- Apache YARN
- Apache HBase
- Apache Kafka
- Neo4j

更多有关 ZooKeeper 的详细信息，请参阅 https://zookeeper.apache.org/ 上的 Apache ZooKeeper 文档。

非卖品，仅供非商业用途或交流学习使用

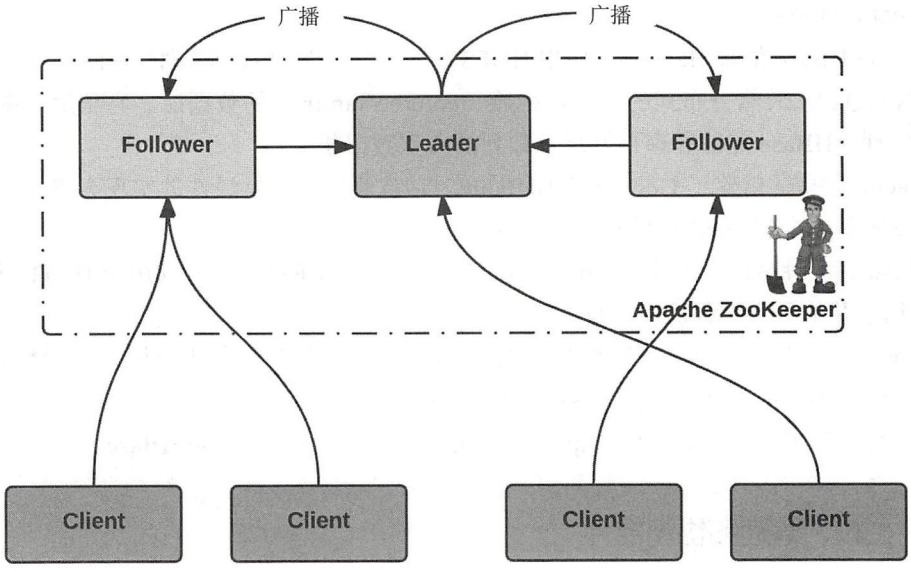


图 9-15 ZooKeeper 的工作原理

2. Apache Oozie

Apache Oozie 是一个用 Java 开发的、开源的 Web 应用程序，用于创建数据流水线，而且能与 Hadoop 技术栈很好地集成。

Oozie 可用于在 Hadoop 集群中调度和运行 Oozie 作业。Oozie 可以将小的作业合并为更复杂的作业，并且可以根据配置的流水线来实现用例需求对应的功能。Oozie 会启动配置好的工作流，并利用 Hadoop 引擎执行工作流中的各个作业。

有两种机制来检测 Oozie 作业的完成情况，分别是回调（callback）和轮询（polling）。在配置作业时，可以配置一个回调的 URL，当作业完成时即调用该 URL。

图 9-16 显示了 Oozie 的基本工作原理。

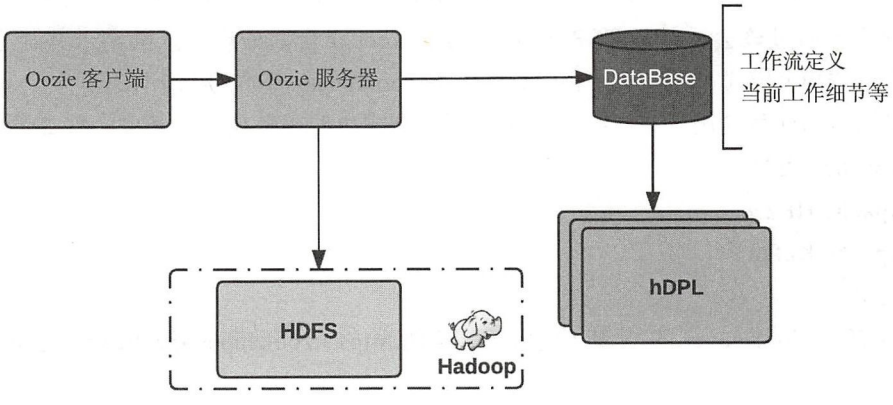


图 9-16 Oozie 的基本工作流程

非卖品，仅供非商业用途或交流学习使用

Oozie 客户端调用服务端将工作流的定义和作业执行细节，以及启动的 Oozie 任务的执行细节都存储在数据库中。该数据库还存储了工作流中所有作业的状态和用于回调的 URL。Oozie 服务器调用 Hadoop 引擎来实际执行作业，并在作业完成时以及完成整个工作流程时接收回调触发器。

更多关于 Apache Oozie 的详细信息，请参阅 http://oozie.apache.org/ 上的 Apache Oozie 文档。

3. Apache Ambari

Apache Ambari 是 Apache Software Foundation 的一个软件项目。通过 Ambari，系统管理员能够调配、管理和监控 Hadoop 集群，并将 Hadoop 与现有的企业基础组件集成起来。Ambari 曾经是 Hadoop 的一个子项目，但现在它已经成为一个顶级项目。

——维基百科

图 9-17 显示了 Apache Ambari 的基本工作原理。

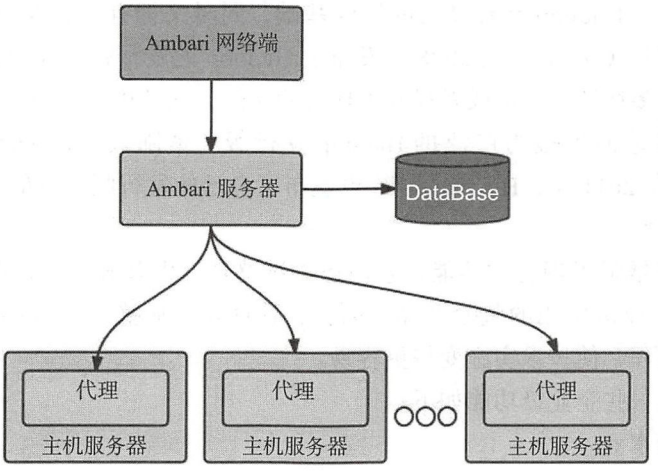


图 9-17　Apache Ambari 架构

Ambari 以客户端—服务器端的模式运行，其中 Ambari 服务器端提供了各种 RESTful 的接口，Ambari 的 Web 应用可以使用这些接口。用户通过这个 Web UI 与 Ambari 进行交互。Hadoop 集群中的每个节点都安装有 Ambari Agent，它将数据发送到 Ambari 服务器中存储起来。Ambari 的监控功能用到了两种开源技术，分别是 Ganglia 和 Nagios。

更多关于 Ambari 的详细信息，请参阅 https://ambari.apache.org/ 上的 Apache Ambari 文档。

9.4.4　数据集成组件

Hadoop 生态系统中能够将 Hadoop 与其他（传统）技术中的数据存储集成起来的组件属于此类。接下来将讨论一些非常知名的组件，并了解它们的工作原理。

非卖品，仅供非商业用途或交流学习使用

1. Apache Sqoop

在第 5 章中，我们已经详细介绍了 Apache Sqoop，这里就不再赘述了。

更多有关 Sqoop 的详细信息，还可以参阅 http://sqoop.apache.org/ 上的 Apache Sqoop 文档。

2. Apache Flume

同样地，第 6 章已经对 Apache Flume 进行了详尽的介绍，这里也不再赘述了。

更多有关 Apache Flume 的详细信息，请参阅 http://flume.apache.org/ 上的 Apache Flume 文档。

9.5 Hadoop 发行版

在开源界，除了数据挖掘和搜索相关的用例外，Hadoop 是首选的大数据处理技术之一。Hadoop 赋予了企业处理组织内部的大数据的能力。

作为开源软件，Hadoop 有着庞大的社区基础，同时又拥有大企业的支持。在 Apache Hadoop 发展的早期，Cloudera 在 2008 年发布了 Hadoop 的发行版。各个发行商都在自己的发行版中增加了许多特性，丰富或者提升了现有的功能，使其更适合用户使用。

Cloudera 仍然是使用最为广泛的 Hadoop 发行版。紧随其后，MapR 于 2009 年发布 Hadoop 发行版，在 2011 年，Hortonworks 也发布了自己的发行版。目前，这三家公司占据了绝大多数市场份额。

这些分布不仅增强了现有的功能，它们还尝试整合许多开源产品，以便企业能采购和使用这些能支持企业级应用的整套产品。而且，这些发行版隐藏了各种版本中的复杂性，同时升级了处理流程，使技术实施变得更容易。

这些发行版的一些企业级功能如下：

- 易于升级版本。
- 确保各个独立的项目协同工作。
- 兼容各种操作系统。
- 企业级支持服务。
- 良好的抽象设计，避免了执行附加脚本和各种 Hadoop 命令的复杂性。
- 及时的 bug 修复。
- 提供了丰富的增强功能来满足企业的需求。
- 提供了关于部署和其他基础组件的咨询支持服务。

如前所述，以下是一些主要的 Hadoop 发行版及其详细信息：

- Cloudera Distribution for Hadoop（CDH）
  - 业界领先。
  - 流行的 Hadoop 发行版。

- 业界第一个丰富和增强了 Hadoop 功能的发行版。
- 为参与 Cloudera Enterprise 订阅服务的企业提供支持。
- 有两个选项可选，分别是免费版和高级版（根据集群的规模）。

- Hortonworks Data Platform（HDP）
  - 完全开源。
  - Amazon 和 IBM 以服务形式提供该发行版。
  - 寻求标准化，并因此成为一段时间内最受支持的发行版。
- MapR Distribution
  - 与 Cloudera 和 Hortonworks 的发行版类似。
  - 集成了自己的数据库系统——MapR DB。
  - 速度快，功能强大。
  - 支持类似 HDFS 的自有文件系统。
  - 有两个选项可选，分别是免费版和高级版。
- Altiscale
  - Hadoop 即服务。
  - 被 SAP 收购。
- Amazon Elastic MapReduce。
  - 基于云的 Hadoop（按需使用）。
  - 与 Apache Hadoop 以及 MapR 的发行版略有不同。

在本节中，读者开源、清晰地了解了市面上的各种 Hadoop 发行版。根据用户组织的需求，用户可以进行明智的抉择，当然也可以选择社区维护的 Hadoop 版本。

9.6　HDFS 和数据格式

Hadoop 将数据存储在大小为 64/128/256 MB 的数据块中。Hadoop 还会检测许多常用文件格式，并在存储时进行相应处理。Hadoop 同样支持数据压缩，虽然压缩方法支持分割和随机访问，但是还是采用了不可分割的数据格式。Hadoop 默认包含多种压缩编解码器（codec）。具体如下：

- 基于文件（file-based）：这与用户在桌面操作系统压缩文件是相似的。有些格式支持分割，有些格式不支持，但其中大部分格式都会保存在 Hadoop 中。这个编解码器压缩整个文件，对任何文件格式都这样操作。
- 基于数据块（block-based）：众所周知，Hadoop 中的数据存储在数据块中，这种编解码器会压缩每个数据块。

但是，压缩会增加 CPU 负载并导致性能降低。Hadoop 支持存储多种传统格式的文件。然而，Hadoop 倾向于一些特定的文件格式，具体如下：

- 文本存储（CSV，Comma Separated Values；TSV，Tab Separated Values；JSON，JavaScript Object Notation 等）：文本文件中每行存储一条数据，并在行尾添加分隔符来划分每条记录。用户还可以使用定义明确的 JSON 格式的记录，并将其存储在 Hadoop 中。当使用这种格式时，通常会使用压缩，因为这些格式天然地支持压缩。
- Avro：一种内置序列化和反序列化能力的文件格式。它可以存储简单或复杂的对象，同时将其中的复杂性进行抽象，并且还提供许多工具给用户，让用户能轻松地管理这些数据。它还支持数据块级别的压缩，同时也是最受欢迎的 Hadoop 文件格式之一。
- 序列文件（sequence file）：专为 MapReduce 设计，因此受到 Hadoop 的广泛支持。每条记录都是一个编码的键值对，同时也支持数据块级别的压缩。
- 列式文件格式（columnar file format）：顾名思义，它在 Hadoop 系统中以水平（行）和垂直（列）方式划分数据，以方便访问数据子集（例如，所有记录中某一列的数据）。如果用户需要查询数据来做幻灯片，对比只有行的文件格式，这种格式可带来更多便利，具体文件格式如下：
  - Parquet：最为广泛使用的列式文件格式。
  - RCFile（Record Columnar File）：它是由 Doug Cutting（Hadoop 的创始人）创建的 Hadoop 中的第一个列式文件格式，具有良好的压缩和性能。
  - ORC File（Optimized RC File）：在 RCFile 基础上加入了压缩和优化，压缩性能比 RCFile 更好。

用户可以根据自身用例的需求选择正确的格式。在进行选择时，用户务必要考虑一些重要的因素，例如读取数据的方式和性能要求。

9.7 Hadoop 用于近实时应用

Hadoop 因其极高的数据批处理计算性能而广受欢迎，即使这些数据具有相当大的差异性和很高的产生速度。但是，近实时应用也有着自身的数据处理需求。

虽然在 Hadoop 生态系统中，Flume 可以提供基于流的数据处理，但是 Hadoop 仍然需要大量自定义的数据处理。Flume 的大部分 Source 和 Sink 的实现都在进行数据 ETL 处理。Flume 实现任何数据处理需求，都需要实现自定义的 Sink 组件。

Spark Streaming 是一种更为成熟的近实时数据处理框架，它与 HDFS 协同工作，按照前文讨论过的微批处理的方式，提供了比 Flume 更为强大的功能，因为 Spark Streaming 是基于数据流水线的近实时处理。

即使有近实时数据处理能力及 HDFS 数据存储，如何从 HDFS 中随机访问数据仍然是一个很大的挑战，因为 HDFS 是一个顺序文件系统。

为了解决数据随机访问的问题，HBase 应运而生，它是 Google 的 Bigtable 架构的一种

实现。虽然 HBase 支持随机访问，但实质上是面向键值对的。只有数据的键可用时才能直接查找该数据。HBase 不适用于任意基于部分匹配的情形，因为这可能导致对 HDFS 内的文件进行扫描。

9.8　Hadoop 部署模式

Hadoop 支持 3 种部署模式，这里对这些模式做简单的介绍：

- Standalone（Local）模式
  - 默认模式。
  - 单 Java（JVM）进程。
  - 非分布式或非集群架构。
  - 三者中最简单的。
  - 几乎只适用于学习和运行范例程序。
  - 非常适合用于调试。
- 伪分布式模式（单节点集群）
  - 单节点安装。
  - Java（JVM）进程中自带守护进程。
  - 可用于在单个节点中模拟多节点集群的安装 / 支持问题（单节点的小型集群安装）。
  - 安装难度显著低于完全分布式安装。
  - 与第一种模式类似，该模式也是一种非分布式安装。
- 完全分布式模式（多节点集群）
  - 完全分布式的集群架构。
  - 几乎生产环境都采用该模式。
  - 根据需求，可以扩展至数千节点。
  - 需要在每个节点上安装 Hadoop 并配置集群信息。
  - 基于 YARN 和自身的功能实现的真正分布式和集群功能。
  - 高度可配置并支持高可用性和安全性。
  - 良好的监控和管理支持（提供各种 Web 界面的组件，用于 Hadoop 的可视化管理）。

用户可以自行选择部署模式。同一份代码可以在任意的部署模式上运行。

9.9　Hadoop 运行范例

本节将结合 SCV 用例，介绍 Apache Hadoop 作为数据存储的完整工作范例。

9.9.1 安装

前面的章节已经介绍了 Hadoop 的伪分布式部署。为了理解本章的例子，我们将继续采用相同的配置。

9.9.2 数据准备

为了展示 Hadoop 示例，我们将进一步扩展客户配置文件中的数据生成工具。对于批处理过程来说，预设的数据量一般来说非常巨大。

因此，我们将尝试为批处理生成约 200 万客户记录及其相关信息。

我们将使用与之前相同的数据生成工具，唯一的区别在于这次将写入 200 万客户记录到数据库中。在关系数据库中生成这么多行数据可能需要一些时间，具体取决于机器配置。

如果读者想在运行数据生成工具之前，清理 PostgreSQL 数据库，那么请按顺序运行以下命令：

```
DELETE from address;
DELETE from customer;
```

在生成数据的 library 所在的目录中运行下列命令，正如第 8 章介绍的一样。

```
java -jar client-generator-1.0-SNAPSHOT.jar config/db.properties  2000000
```

现在，已经在数据库中创建了 200 万条记录，然后再生成 200 万条客户联系方式信息，唯一的区别是这部分信息将为逗号分隔的文本文件的格式。在生成数据的 library 所在的目录中运行下列命令，正如第 8 章介绍的一样。

```
java -jar web-generator-1.0-SNAPSHOT.jar contacts.log 2000000 contact
```

上一条命令生成的文本文件大致的模式如下所示：

cust_id:int	cell_number:string	work_phone:string	email_address:string

在这些例子中，我们还希望能够将数据可视化为 Hive 表，因此还需要安装和配置 Hive 服务。作为 Apache 生态系统的一部分，前文中对 Apache Hive 已经有了大致的介绍。

9.9.3 安装 Hive

1）使用以下命令下载 Hive 的最新稳定版本：

```
wget
http://www-us.apache.org/dist/hive/hive-2.1.1/apache-hive-2.1.1-bin
.tar.gz
```

2）切换到用户目录并使用以下命令提取 tar 包中的内容：

```
tar -xzvf ${DOWNLOAD_DIR}/apache-hive-2.1.1-bin.tar.gz
```

3）将解压目录配置到环境变量 ${HIVE_HOME} 中，并将其中的二进制文件添加到该路径中。将下列命令追加到 ~/.bashrc 文件中。

```
export HIVE_HOME=<Hive directory>
export PATH=$PATH:$HIVE_HOME/bin
```

4）安装最新的 SASL（Simple Authentication and Security Layer），因为 Hive 需要依赖它。对于 CentOS，可以使用以下命令安装这些软件：

```
sudo yum install *sasl*
```

5）Hive 服务需要元数据信息管理的数据存储。在 shell 终端中使用 psql（与 PostgreSQL 配合使用的交互式终端）客户端，使用以下命令将 PostgreSQL 服务器配置为元数据服务：

- 使用以下命令创建 hiveuser 作为 postgresql 用户，并创建数据库作为元数据存储：

```
sudo -u postgres psql
```

上述命令会初始化并启动 psql 客户端（shell 提示符为 postgres = # ）用来运行查询。然后使用以下查询为 Hive 的 metastore 创建用户和数据库：

```
 postgres=# CREATE USER hiveuser WITH PASSWORD
'mypassword';
 postgres=# CREATE DATABASE metastore;
```

- 使用以下命令配置 hiveuser 的 metastore 的权限：

```
postgres=# \c metastore
metastore=# \pset tuples_only on
metastore=# \o /tmp/grant-privs
metastore=#   SELECT 'GRANT SELECT,INSERT,UPDATE,DELETE ON
"'  || schemaname || '". "' ||tablename ||'" TO hiveuser ;'
metastore-#   FROM pg_tables
metastore-#   WHERE tableowner = CURRENT_USER and
schemaname = 'public';
metastore=# \o
metastore=# \pset tuples_only off
metastore=# \i /tmp/grant-privs
```

- 使用如下命令将 ${HIVE_HOME}/conf/hive-default.xml.template 文件复制到 ${HIVE_HOME}/conf/hive-site.xml。

```
cp ${HIVE_HOME}/conf/hive-default.xml.template
${HIVE_HOME}/conf/hive-site.xml
```

- 在 ${HIVE_HOME}/conf/hivesite.xml 文件中配置表 9-1 所示属性。

表 9-1　在 ${HIVE_HOME}/conf/hivesite.xml 文件中配置属性

Hive 属性	建 议 值
hive.exec.scratchdir	/tmp/hive
hive.exec.local.scratchdir	/tmp/hive/centos 这里配置的 centos 就是执行 hive 查询的账户

仅供非商业用途或交流学习使用

（续）

Hive 属性	建 议 值
hive.downloaded.resources.dir	/tmp/hive/${hive.session.id}_resources
javax.jdo.option.ConnectionPassword	hivepass psql 客户端中创建的密码
javax.jdo.option.ConnectionURL	jdbc:postgresql://<POSTGRESQL_SERVER_IP:PORT>/metastore
javax.jdo.option.ConnectionDriverName	org.postgresql.Driver
javax.jdo.option.ConnectionUserName	hiveuser
hive.server2.enable.doAs	false

- 按照与 Sqoop 相同的设置方式，将 PostgreSQL 驱动的 JAR 文件复制到 Hive 的 lib 目录，即 ${HIVE_HOME}/LIB。使用以下命令下载 PostgreSQl 的驱动。作为 Sqoop 设置的一部分，该驱动应该已经存在于下载文件夹中。

```
wget
https://jdbc.postgresql.org/download/postgresql-9.4.1212.jr e6.jar
```

6）在 ${HADOOP_HOME}/etc/hadoop/core-site.xml 文件中加入下列配置。将 hadoop.proxyuser.centos 配置为用户账户，替换掉已有的 centos，按照如下配置示例：

```
<property>
   <name>hadoop.proxyuser.centos.hosts</name>
   <value>*</value>
</property>
<property>
   <name>hadoop.proxyuser.centos.groups</name>
   <value>*</value>
</property>
```

7）使用以下命令重新启动 dfs 服务：

```
stop-dfs.sh
start-dfs.sh
```

8）在 ${HUE_HOME}/desktop/conf/hue.ini 文件中配置下列属性，使得 Hue 可以与 Hive 服务协同工作。在 hue.ini 中搜索下列属性并将 centos 改成用户的账户名。

```
hive_conf_dir=/home/centos/apache-hive-2.1.1-bin/conf
```

9）重新启动 Hue 服务，先妥善地退出（找到监控进程并杀死该进程），再用以下命令启动：

```
${HUE_HOME}/build/env/bin/supervisor -d
```

10）使用以下命令，用 schematool 生成模式：

```
${HIVE_HOME}/bin/schematool -dbType postgres -initSchema --verbose
```

仅供非商业用途或交流学习使用

进入 pgAdmin，可以看到一个新的数据库，即 metastore，其中包含了 schematool 生成的多个表格。

11）按照以下步骤验证安装：

① 执行以下命令来检查 Hive shell：

```
${HIVE_HOME}/bin/hive
```

如果顺利，用户可以看到 Hive shell 的界面。根据需求，用户可以使用该 shell 运行各种 Hive 查询。

② 用下面的命令启动 hiveserver2 服务。

hiveserver2 是一个远程进程，通过它 Hue 可以与 Hive 集成，还可以在 Hue 的 UI 中运行查询。

```
${HIVE_HOME}/bin/hive --service hiveserver2 -hiveconf
hive.root.logging=console
```

③ hiveserver2 成功启动后，打开 Hue 并跳转到 QueryEditor|HIVE，此处应该没有任何报错。

现在，已经安装了所有必需的组件，之后会介绍几个示例。最初的例子涉及数据加载方面，随着示例的继续运行，还将看到 Hadoop 层的数据处理方面。

9.9.4　范例——批量数据加载

现在，我们将介绍如何批量加载数据到 Hadoop 层中，并介绍 Hadoop 生态系统中对此的支持。

1. 文件数据加载

1）使用 fs put 命令，可以很容易地将 Linux 服务器的文件复制到 HDFS 集群中。Hadoop 客户端中包含该命令，而 Hadoop 客户端可以安装到任何 Linux 操作系统中。在本书的例子中，Hadoop 客户端作为 Hadoop 伪分布式部署的一部分。

该命令的一般语法如下：

```
hdfs dfs -put /local/path/test.file
hdfs://namenode:9000/user/stage
```

2）该范例在 HDFS（HDFS 的某个文件夹中）中创建了一个原始数据区域。使用下列命令来创建这个区域，该区域用于存储原始格式的数据。

```
hdfs dfs -mkdir -p /<any-path>/raw/txt
```

执行完上述命令后，将在 HDSF 中创建文件夹结构（<anypath>/raw/txt），可以在 NameNode 的 UI 中查看该文件夹。

3）切换到生成联系人信息文件的目录，然后运行以下命令：

仅供非商业用途或交流学习使用

```
hdfs dfs -put  contacts.log hdfs://<hadoop-namenode-ip-
address>:9000/<any-path>/raw/txt/contact.log
```

4）现在可以在 Hue 中查看该文件的内容，如图 9-18 所示。

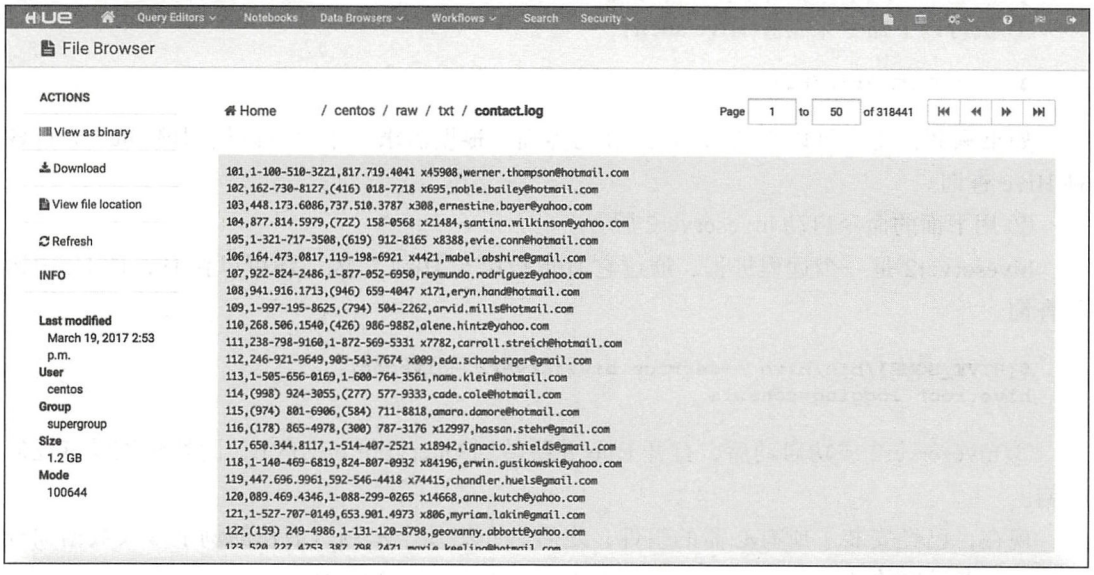


图 9-18　通过 dfs put 将文本数据加载到 HDFS 中

2. RDBMS 数据加载

正如前文中第 5 章中介绍的那样，可以使用 Sqoop 从 RDBMS 批量加载数据。可利用 Sqoop 相关章节中介绍的命令来实现这个需求。

9.9.5　范例——MapReduce 数据处理

现在开始使用 MapReduce 过程加载 contacts.log 文件中的数据。可以用一个非常简单的 MapReduce 程序执行数据加载操作，如下文的代码片段所示。本章所有的源代码都保存在 GitHub 上仓库的 chapter09 文件夹中。chapter09/contacts-loader-mr 工程中包含下列代码块：

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import java.io.IOException;
import java.util.Random;

public class ContactsLoader  {
   public static void main(String[] args) throws Exception {
      //定义Job Configurations
```

仅供非商业用途或交流学习使用

```
        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "Contacts Loader - Simple Mapper");
        job.setJarByClass(ContactsLoader.class);
        job.setMapperClass(LoadMapper.class);
        //包括对file:// 和hdfs:// file schemes的支持
        conf.set("fs.hdfs.impl",
            org.apache.hadoop.hdfs.DistributedFileSystem.class.getName()
        );
        conf.set("fs.file.impl",
            org.apache.hadoop.fs.LocalFileSystem.class.getName()
        );
        //设置输入和输出路径
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
    private static class LoadMapper extends Mapper<Object, Text,
        LongWritable, Text> {
        @Override
        protected void setup(Context context) throws IOException,
            InterruptedException {
            super.setup(context);
        }
        @Override
        protected void map(Object key, Text value, Context context) throws
            IOException, InterruptedException {
            String line = value.toString();
            try {
                context.write(new LongWritable(Long.parseLong(line.substring(0,
                    line.indexOf(",")))), value);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

切换到 chapter09/contacts-loader-mr 目录，并运行以下命令来编译此代码：

```
mvn install
```

编译完成后，会得到一个可以执行的 JAR 文件，它包含了所有的依赖文件，之后就可以启动 MapReduce 作业了。需要使用如下 maven POM 文件配置，包括 shaded 插件（用于生成可执行 JAR 文件）。这个 POM 文件包含了少量 Hadoop 依赖，在 build 域中配置了 shaded 插件，用 ManifestResourceTransformer 声明主类，并使用 ServiceResourceTransformer 去掉不需要的文件。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
<parent>
<groupId>com.laketravels</groupId>
<artifactId>chapter09</artifactId>
<version>1.0-SNAPSHOT</version>
</parent>

<groupId>com.laketravels.batch</groupId>
<artifactId>contacts-loader-mr</artifactId>
<packaging>jar</packaging>

<dependencies>
<dependency>
<groupId>org.apache.hadoop</groupId>
<artifactId>hadoop-client</artifactId>
<version>2.7.3</version>
</dependency>

<dependency>
<groupId>org.apache.hadoop</groupId>
<artifactId>hadoop-hdfs</artifactId>
<version>2.7.3</version>
</dependency>
</dependencies>
<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-shade-plugin</artifactId>
<version>3.0.0</version>
<executions>
<execution>
<phase>package</phase>
<goals>
<goal>shade</goal>
</goals>
<configuration>
<transformers>
<transformer
implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTra
nsformer">
<mainClass>com.laketravels.ch09.batch.mr.loader.ContactsLoader</mainClass>
</transformer>
<transformer
implementation="org.apache.maven.plugins.shade.resource.ServicesResourceTra
nsformer"/>
</transformers>
</configuration>
</execution>
</executions>
</plugin>
</plugins>
</build>
</project>
```

切换到生成 JAR 文件的 contacts-loader-mr 工程，并使用以下命令运行：

```
java -jar target/contacts-loader-mr-1.0-SNAPSHOT.jar file:///<any-
```

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用


```
path>/contacts.log hdfs://<hadoop-namenode-ip-address>:9000/<any-
path>/raw/contact/mr
```

为了实现可视化和查询功能，还可以在底层数据上创建 Hive 表。Hive 有一个 EXTERNAL 表的概念，它可以将文件装载到 Hive 表。这对于数据湖的实施非常有帮助，因为数据可以持续写入文件而 Hive 表也可以反映出新增的数据。在本书后面的范例中，读者将看到 Hive 使用内置的 SerializerDeserializer（SerDe）支持 Avro 和 Parquet 数据存储。

1. 文本数据转存为 Hive 表

本节将可视化展示之前批量加载数据到 Hive 表中的输出，如同前文中的 MapReduce 示例程序处理过程。

为了实现这一点，需要按照给定的步骤操作：

1）打开 Hue 并跳转到 Query Editor|HIVE。

2）在 Query 框中，执行一个 DDL 命令，基于前面上传的存储联系人信息的文件之上创建一个 Hive 表，如表 9-2 所示。

表 9-2　创建 Hive 表

使用 Hive 表进行文本文件可视化	MapReduce 输出到 Hive 表
CREATE EXTERNAL TABLE IF NOT EXISTS ContactsText(id STRING, cell STRING, phone STRING, email STRING) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' STORED AS TEXTFILE location '<hdfs-path-to-txt-file>';	CREATE EXTERNAL TABLE IF NOT EXISTS ContactsMR(id STRING, cell STRING, phone STRING, email STRING) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' STORED AS TEXTFILE location '<hdfs-path-to-mr-upload-data>';

3）一旦 Hive 返回成功的消息，就运行 select 查询来验证数据，如图 9-19 中的命令所示。

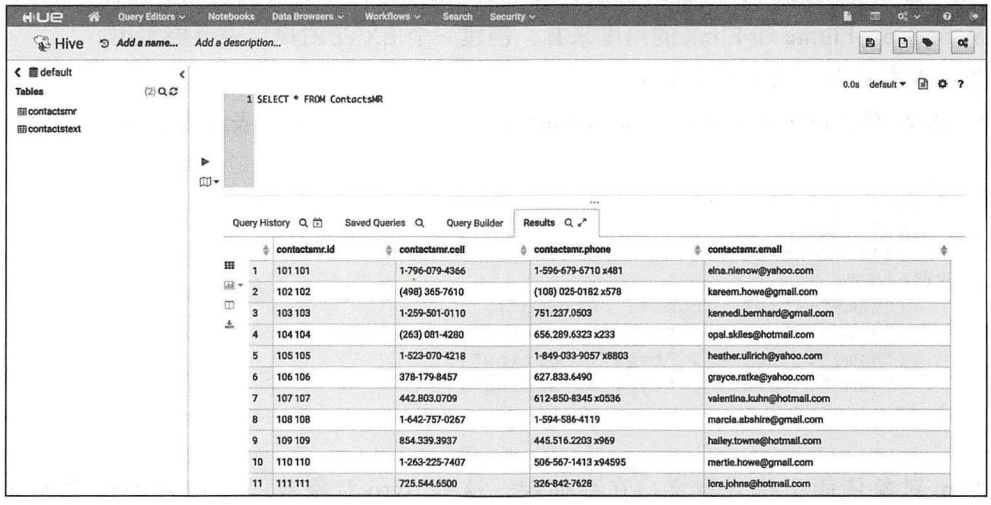


图 9-19　文本文件的外部表视图

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

观察上面的结果集，数据已经在 Hive 表中正确显示。如图 9-20 所示，在 Hive UI 中可以看到，MapReduce 任务上传的数据在 contactsMR 表中已有正确的反映。

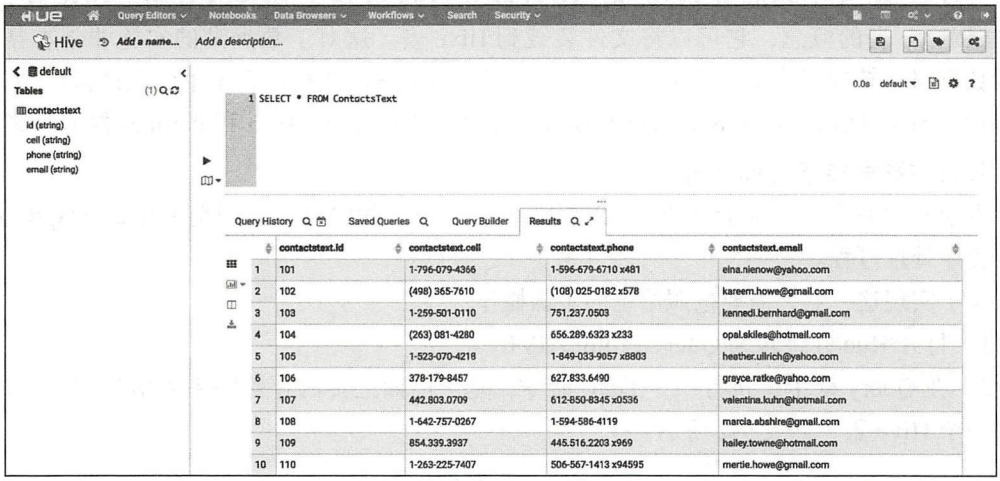


图 9-20 通过 MapReduce 加载文本文件到 HDFS 中

正如读者看到的，由于 Hadoop 文件是一个用键值对元组组成的顺序文件，因此联系人的 id 列看起来有问题。该 id 出现了两次（范例记录：101 101，1-796-079-4366，1-596-679-6710 x481，elna.nienow@yahoo.com），这是因为使用了完整的联系人记录的数据元组作为值，其中也包含了 id。

2. Avro 数据转存 Hive 表

为了使用包含了联系人信息的文本文件创建 Avro 格式数据，需要使用 Hive 的 INTERNAL 表。Hive 具有内置的机制将文本数据转换为 Avro 格式数据。然而，如前文所述，从 Sqoop、Flume 和 Flink 的角度来看，创建一个 EXTERNAL 表会更实用。

为了查看效果，接下来需要执行以下步骤：

1）Avro 格式的联系人数据，即 contact.ascv（模式文件）可以表示为如下所示的命令：

```
{
   "namespace": "example.avro",
   "type": "record",
   "name": "Contact",
   "fields": [
      {"name": "id", "type": "string"},
      {"name": "cell",  "type": "string"},
      {"name": "phone", "type": "string"},
      {"name": "email", "type": "string"}
   ]
}
```

Avro 对象依赖于格式定义，在存储层，这些 Avro 对象被序列化为 Avro 数据文件。Avro 序列化器需要引用此模式才能执行序列化。Avro 序列化应该天然是增量的，以便基于

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

Avro 数据文件创建外部表。当使用 Sqoop 作业将加载 Avro 文件数据时，Sqoop 会生成一个默认模式用于序列化 Avro 数据文件。

如前所述，Avro 模式的定义与 JSON 模式规范草案类似。但是，Avro 模式声明在支持的数据类型和数据结构方面存在差异。

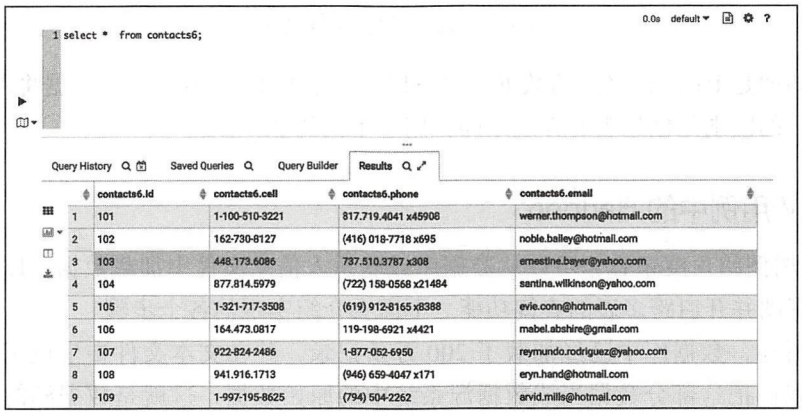
2）在 Hive 编辑器中，运行以下命令来创建另一个 Hive 表，但这次是使用内联模式定义的 Avro 数据格式：

```
CREATE TABLE contactsAvro
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'
STORED AS INPUTFORMAT
'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'
OUTPUTFORMAT
'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'
TBLPROPERTIES (
   'avro.schema.literal'='{"namespace": "example.avro",
   "type": "record",
   "name": "Contact",
   "fields": [
         {"name": "id", "type": "string"},
         {"name": "cell",  "type": "string"},
         {"name": "phone", "type": "string"},
         {"name": "email", "type": "string"}
      ]
   }
');
```

3）在 Hive Query Builder 中使用 INSERT OVERWRITE 查询语句加载数据到此 Hive 表中。这可能需要花些时间，因为它会在内部启动 MapReduce 作业来加载这些数据：

```
INSERT OVERWRITE TABLE contactsAvro SELECT id, cell, phone, email
FROM contactsText;
```

4）从新创建的表（contactsAvro）中查询，得到与之前相同的结果，但是如果查询 Hive 仓库，将看到数据加载操作创建的 Avro 数据文件，如图 9-21 所示。



	contacts6.id	contacts6.cell	contacts6.phone	contacts6.email
1	101	1-100-510-3221	817.719.4041 x45908	werner.thompson@hotmail.com
2	102	162-730-8127	(416) 018-7718 x695	noble.bailey@hotmail.com
3	103	448.173.6086	737.510.3787 x308	ernestine.bayer@yahoo.com
4	104	877.814.5979	(722) 158-0568 x21484	santina.wilkinson@yahoo.com
5	105	1-321-717-3508	(619) 912-8165 x8388	evie.conn@hotmail.com
6	106	164.473.0817	119-198-6921 x4421	mabel.abshire@gmail.com
7	107	922-824-2486	1-877-052-6950	reymundo.rodriguez@yahoo.com
8	108	941.916.1713	(946) 659-4047 x171	eryn.hand@hotmail.com
9	109	1-997-195-8625	(794) 504-2262	arvid.mills@hotmail.com

图 9-21　支持 Arvo 的 Hive 表——利用 INSERT OVERWRITE 语句加载数据

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

图 9-22 显示了 Hive warehouse 目录下的 Avro 数据文件。

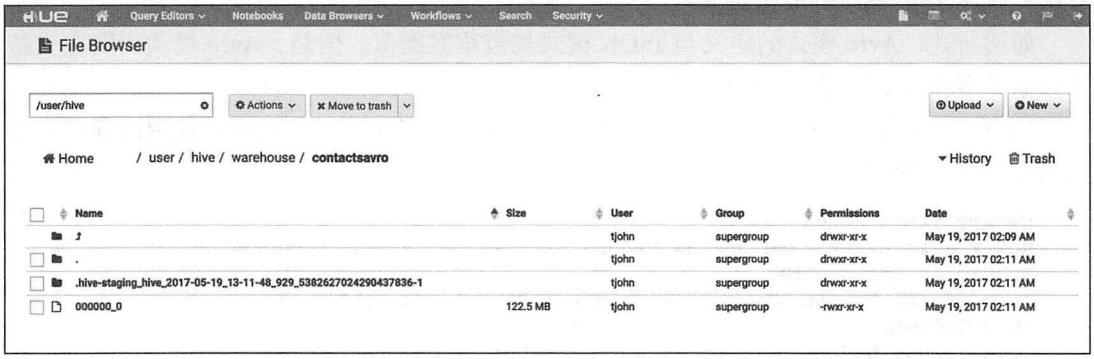


图 9-22 生成的 Avro 数据文件

图 9-23 显示了其中一个 Avro 数据文件的内容。

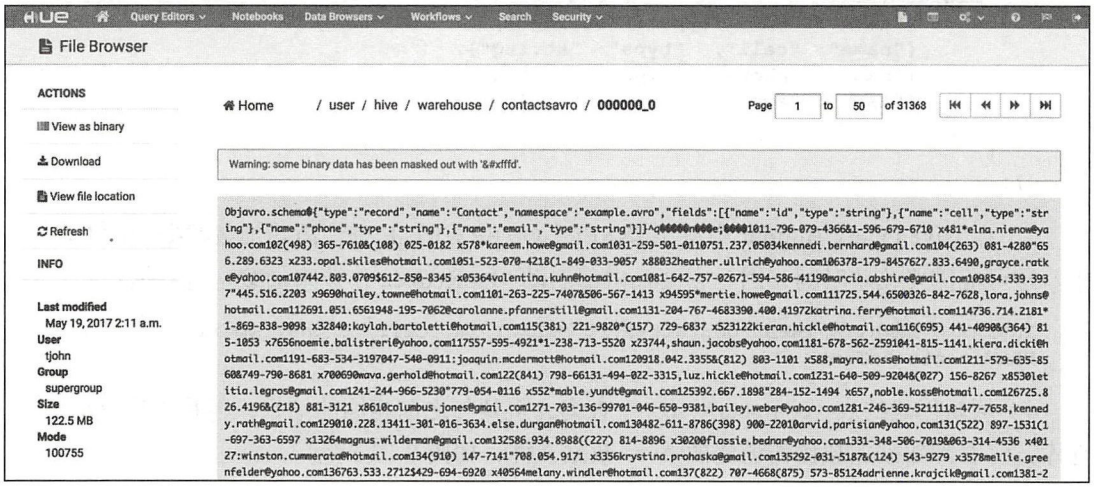


图 9-23 Avro 数据文件视图

同样，即便是 Parquet 格式的数据，也可以为之定义 Hive 表，其中数据将以 Parquet 格式存储。唯一的区别是创建表的方式不同，用了不同的 SERDE。

9.9.6 SCV 用例中的 Hadoop

从 SCV 用例的角度来看，可以从数据库或联系人信息文件中加载数据，以便于向用户提供完整的视图并开启跨多表的查询功能。下面会介绍实现的各个步骤。

如前文所述，数据库中已经加载了 200 万条记录，并在文本文件中生成了 200 万个联系人信息。我们把这部分数据当成数据源系统中的原始数据，这就是数据湖的原始数据存储区。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

1. 初始目录设置

运行以下命令，在 HDFS 中创建下列目录：

```
hdfs dfs -mkdir -p /datalake/raw/customer
hdfs dfs -mkdir -p /datalake/raw/address
hdfs dfs -mkdir -p /datalake/raw/contacts
```

2. 数据加载

现在，从数据库中加载数据，每次加载数据库中的一张表，然后存储在不同 RAW 存储区中：

1）加载用户数据：

① 使用以下命令，运行一个 Sqoop 作业从数据库中导入用户属性数据文件：

```
${SQOOP_HOME}/bin/sqoop import --connect
jdbc:postgresql://<DB-SERVER-
ADDRESS>/sourcedb?schema=public --table customer --m 10 --
username postgres --password <DB-PASSWORD> --as-
avrodatafile --append --target-dir /datalake/raw/customer
```

② Sqoop 的 MapReduce 作业完成后，customer 目录中填充了许多 Avro 数据文件，如图 9-24 所示。

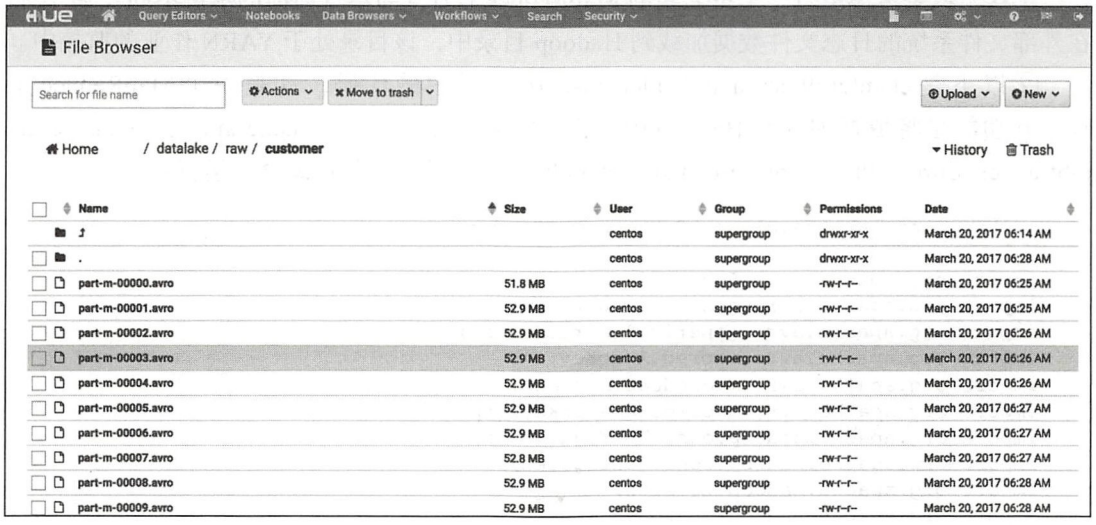


图 9-24　通过 Sqoop 加载客户信息（存储为 Avro 格式）

2）加载用户地址数据：

① 使用以下命令，运行一个 Sqoop 作业从数据库中导入用户地址数据文件：

```
${SQOOP_HOME}/bin/sqoop import --connect
jdbc:postgresql://<DB-SERVERADDRESS>/sourcedb?schema=public
--table address --m 10 --username postgres --password <DB-
```

非卖品，仅供非商业用途或交流学习使用


```
PASSWORD> --as-avrodatafile --append --target-dir
/datalake/raw/address
```

② Sqoop 的 MapReduce 作业完成后，地址目录中填充了许多 Avro 数据文件，如图 9-25 所示。

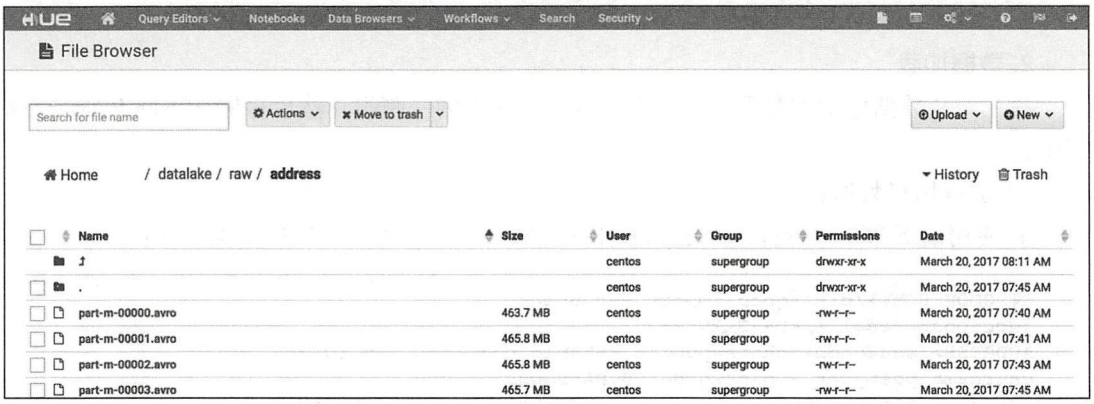


图 9-25 通过 Sqoop 加载客户地址信息（存储为 Avro 格式）

3）从日志文件加载联系人数据：

①这里需要考虑运行一个更完整的 MapReduce 程序（与本章前面的示例不同），将存储在外部文件系统的日志文件数据加载到 Hadoop 目录中，该目录处于 YARN 作业的监控中。

②以下是 chapter09/contacts-loader-mr-avro 工程中的代码，这是一个 MapReduce 工程，其功能是将数据加载到 Hadoop 中。首先需要根据需求更改 fs.defaultFS、mapreduce.jobtracker.address 和 yarn.resourcemanager.address 等配置项，然后编译并运行：

```
package com.laketravels.ch09.batch.mr.loader;

import org.apache.avro.Schema;
import org.apache.avro.generic.GenericData;
import org.apache.avro.generic.GenericRecord;
import org.apache.avro.mapred.AvroKey;
import org.apache.avro.mapreduce.AvroJob;
import org.apache.avro.mapreduce.AvroKeyOutputFormat;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import
org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import
org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import
```

非卖品，仅供非商业用途或交流学习使用


```
org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

import java.io.IOException;


public class ContactsLoader extends Configured implements
Tool{

public static void main(String[] args) throws Exception {
int exitCode= ToolRunner.run(new ContactsLoader(),args );
        System.out.println("Exit code "+exitCode);
    }

public int run(String[] args) throws Exception {
        Configuration conf = new Configuration();

// 集成YARN端点，用于作业控制与监测
        conf.set("fs.defaultFS",
"hdfs://192.168.0.117:9000");
        conf.set("mapreduce.jobtracker.address",
"192.168.0.117:54311");
        conf.set("mapreduce.framework.name", "yarn");
        conf.set("yarn.resourcemanager.address",
"192.168.0.117:8032");

// 支持file://及hdfs://文件模式
        conf.set("fs.hdfs.impl",
org.apache.hadoop.hdfs.DistributedFileSystem.class.getName(
)
        );
        conf.set("fs.file.impl",
org.apache.hadoop.fs.LocalFileSystem.class.getName()
        );

        Job job= Job.getInstance(conf,"Contacts Loader");
        job.setJarByClass(ContactsLoader.class);

// 设置输入/输出路径
        FileInputFormat.setInputPaths(job, new
Path(args[0]));
        FileOutputFormat.setOutputPath(job, new
Path(args[1]));

// 为Avro数据加载配置Schema
        Schema.Parser parser = new Schema.Parser();
        Schema
schema=parser.parse(Thread.currentThread().getContextClassL
oader()
.getResourceAsStream("contacts.avsc"));

// 覆盖默认的类加载策略以加载特定用途的库
        job.getConfiguration().setBoolean(
                Job.MAPREDUCE_JOB_USER_CLASSPATH_FIRST,
true);
```

```
// 配置执行作业的Mapper、Reducer类
        job.setMapperClass(LoadMapper.class);
        job.setReducerClass(LoadReducer.class);


// 设置Map输入/输出值格式，以及处理最终输出的Format类
        job.setMapOutputValueClass(NullWritable.class);
        job.setInputFormatClass(TextInputFormat.class);
job.setOutputFormatClass(AvroKeyOutputFormat.class);

// 设置Avro Job的Schema参数，用于处理Map、Reduce输出
        AvroJob.setMapOutputKeySchema(job, schema);
        AvroJob.setOutputKeySchema(job, schema);

int status = job.waitForCompletion(true) ? 0 : 1;
return status;
    }

private static class LoadMapper extends


Mapper<LongWritable, Text, AvroKey<GenericRecord>,
NullWritable> {
        Schema schema;

@Override
        protected void setup(Context context) throws
IOException, InterruptedException {
super.setup(context);
//为 contact 设置 Schema 初始化
            Schema.Parser parser = new Schema.Parser();
schema=parser.parse(Thread.currentThread().getContextClassL
oader()
                    .getResourceAsStream("contacts.avsc"));
        }


@Override
        protected void map(LongWritable key, Text value,
Context context) throws IOException, InterruptedException {
try {
// 用于记录人口信息的 Schema 与数据
                GenericRecord record = new
GenericData.Record(schema);
                String inputRecord=value.toString();
                String[] values = inputRecord.split(",");
                record.put("id", values[0]);
                record.put("cell", values[1]);
                record.put("phone", values[2]);
                record.put("email", values[3]);

                context.write(new AvroKey(record),
NullWritable.get());
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
```

```
    }

private static class LoadReducer extends
Reducer<AvroKey<GenericRecord>,NullWritable,AvroKey<Generic
Record>,NullWritable> {

@Override
        public void reduce(AvroKey<GenericRecord> key,
Iterable<NullWritable> value, Context context) throws
IOException, InterruptedException{
  try {
                context.write(key, NullWritable.get());
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
  }
```

③代码内有注释用来帮助理解源代码。在源码中改好配置后，切换到 chapter09/contacts-loader-mr-avro 目录，执行以下命令进行编译：

```
mvn install
```

④先确保 YARN 已经安装正确。为了配置 YARN，需要在 ${HADOOP_HOME} /etc/hadoop/yarn-site.xml 中添加以下配置：

```
<property>
  <name>yarn.nodemanager.aux-services</name>
  <value>mapreduce_shuffle</value>
</property>
<property>
  <name>yarn.nodemanager.aux-
services.mapreduce_shuffle.class</name>
  <value>org.apache.hadoop.mapred.ShuffleHandler</value>
</property>
```

⑤运行以下命令启动 YARN 进程：

```
${HADOOP_HOME}/sbin/start-yarn.sh
```

执行 jps 命令应该能看到 NodeManager 和 ResourceManager 进程。

⑥前面的代码可以用下面的 YARN 命令来运行：

```
yarn -jar target/contacts-loader-mr-avro-1.0-SNAPSHOT.jar
file:///<any-path>/contacts.log hdfs://<hadoop-namenode-ip-
address>:9000/datalake/raw/contacts/load1
```

⑦下面的 YARN URL 页面可以监控该作业：

```
http://<YARN_BINDING_ADDRESS>:8088
```

图 9-26 显示了 YARN UI 中正在运行的应用程序。

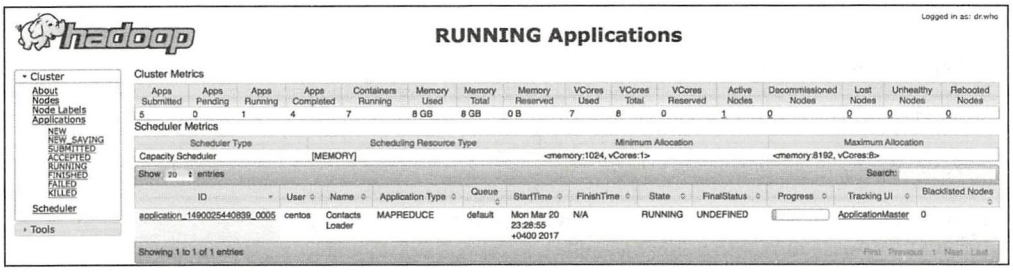


图 9-26 在 YARN UI 中查看应用程序状态

图 9-27 显示了 YARN UI 中的调度器（Scheduler）。

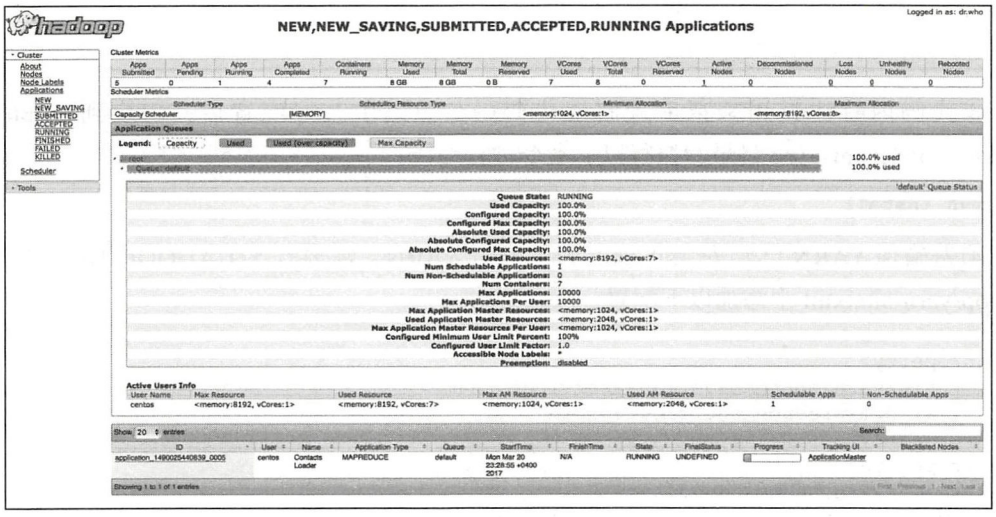


图 9-27 在 YARN UI 中查看调度器状态

⑧作业执行完后，会生成 Avro 数据文件，如图 9-28 中 Hue UI（HDFS 浏览器）所示。

图 9-28 在 Hue UI 中查看通过 MapReduce 加载的数据

3. 通过 Hive 表进行数据可视化

下面通过执行 Hive 脚本（使用 HiveQueryEditor）在二进制 Avro 格式的数据之上创建

Hive 表，从而将加载的数据进行可视化展示：

1）执行以下脚本创建用客户信息（customer）Hive 表：

```
CREATE EXTERNAL TABLE customer
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'
STORED AS INPUTFORMAT
'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'
OUTPUTFORMAT
'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'
LOCATION '/datalake/raw/customer'
TBLPROPERTIES (
  'avro.schema.literal'='{"namespace": "example.avro",
    "type": "record",
    "name": "Customer",
    "fields": [
       {"name": "id", "type": "int"},
       {"name": "first_name",  "type": "string"},
       {"name": "last_name", "type": "string"},
       {"name": "dob", "type": "long"}
    ]}'
);
```

2）执行以下脚本来创建地址（address）Hive 表：

```
CREATE EXTERNAL TABLE address
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'
STORED AS INPUTFORMAT
'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'
OUTPUTFORMAT
'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'
LOCATION '/datalake/raw/address'
TBLPROPERTIES (
  'avro.schema.literal'='{"namespace": "example.avro",
    "type": "record",
    "name": "Address",
      "fields": [
        {"name": "id", "type": "int"},
        {"name": "street1",  "type": "string"},
        {"name": "street2", "type": "string"},
        {"name": "city", "type": "string"},
        {"name": "state", "type": "string"},
        {"name": "country", "type": "string"},
        {"name": "zip_pin_postal_code", "type": "string"}
    ]}'
);
```

3）通过执行以下脚本创建联系人信息（contact）Hive 表：

```
CREATE EXTERNAL TABLE contacts
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'
STORED AS INPUTFORMAT
'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'
OUTPUTFORMAT
'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'
LOCATION '/datalake/raw/contacts/load1'
TBLPROPERTIES (
```

```
    'avro.schema.literal'='{"namespace": "example.avro",
      "type": "record",
      "name": "Contact",
      "fields": [
        {"name": "id", "type": "string"},
        {"name": "cell",  "type": "string"},
        {"name": "phone", "type": "string"},
        {"name": "email", "type": "string"}
      ]}'
);
```

现在所有的数据已经输入到 Hadoop 中，并且表示为 Hive 的外部表，这些表的实际数据来源不同，却都汇聚到数据湖中，并且还能进行查询以及进一步处理。加上本章介绍的存储机制，数据湖的整体视图已经跃然纸上了，如图 9-29 所示。

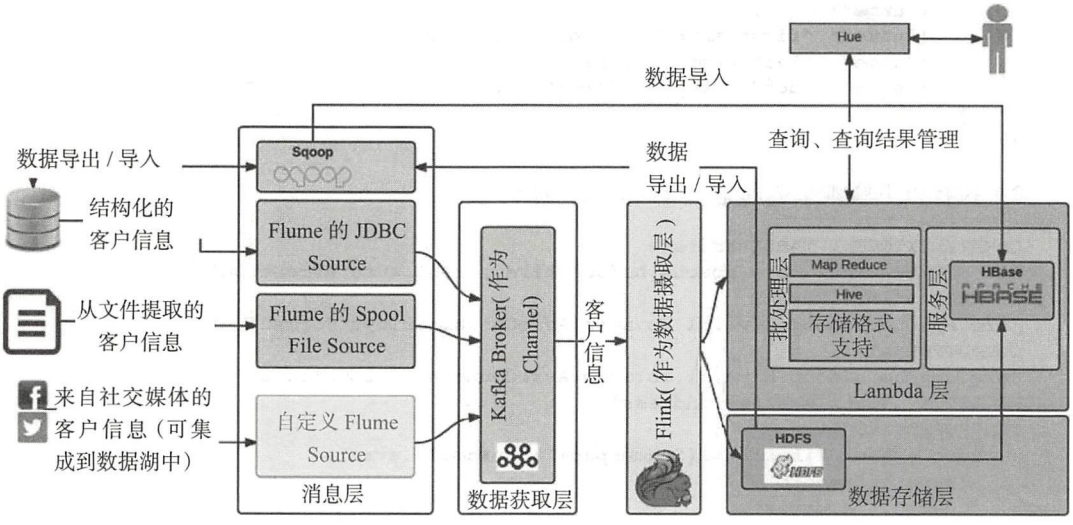


图 9-29 数据湖的单一客户视图

9.10 不适合使用 Hadoop 的场景

并非所有用例都需要 Hadoop，在不适合的场景中强行使用 Hadoop 可能会适得其反。遇到下列需求时不应该使用 Hadoop：

- 处理基于图的数据。Hadoop 生态系统的其他产品（比如 Apache Tez）可能能支持这类需求。
- 处理实时数据。虽然 Hadoop 生态系统中的许多产品能支持这类需求，但是必须充分分析后再决定。基于 HDFS 的 Apache Flink 或者 Spark 是一个值得考虑的选项。
- 处理存储在关系型数据库中的数据。Hive 是一个值得考虑的选项，而 HDFS 不能胜任。
- 数据处理过程中需要共享状态的情况。Hadoop 的工作是通过在集群中的多个节点之间分割数据来完成，然后以并行方式执行作业，本质上是无状态的。

- 处理小数据集。

9.11　其他选项

任何时候使用大数据术语时，用户总会想到 Apache Hadoop。在处理大数据时，Hadoop 几乎是不可或缺的部分。毫无疑问，Hadoop 是一个出色的选项，但是在必须进行技术选型时，Hadoop 的一些内在问题会受到开发人员的质疑，尤其是在企业的大数据处理能力不断提升的背景下，这显然是由于业务的动态变化。Hadoop 的缺点来源于它的复杂性以及实际执行方式。由于这些原因，近来出现一些简化 Hadoop 的创新产品，比如 Pig 脚本和 Apache Spark。

Pig 脚本使用 Pig Latin 语言简化了 MapReduce 活动，同时仍然支持非 Java 开发人员通过简单的编程来执行 MapReduce。

另一方面，Apache Spark Streaming 使用多种编程语言（如 Scala、Java 和 Python）简化了查询机制。在我们的例子中可以看到，Hive 适合查询 HDFS 数据，但是当涉及表联接查询时，它很可能触发一个更复杂的 MapReduce 作业，且失败概率较高。因此，Hive MapReduce 已经被弃用，因为用 Apache Spark 和 Apache Tez 能够以大幅优化的方式完成相同的功能。

9.12　总结

在本章中，与其他章节类似，首先详细讨论了 Apache Hadoop 在数据湖中所处的层次。然后把这项技术，即 Hadoop，引入该层次中讨论，最后详细介绍该项技术。

本章首先给出了选择 Hadoop 的理由，然后介绍这项技术的历史，包括其优势和劣势。紧接着，深入探讨了 Hadoop 内部机制，分别介绍了 Hadoop 1.x 和 Hadoop 2.x 的架构特点。由于本书选择使用 Hadoop 2.x，因此针对性地介绍了构成 Hadoop 2.x 的组件。之后，介绍了 Hadoop 生态系统中的重要组件，这些组件会部分应用到 SVC 用例中。

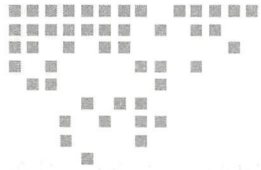
在这之后还深入研究了 Hadoop 的其他方面，包括 Hadoop 发行版、HDFS、各种文件格式，最后还介绍了各种部署模式。此外，还深入研究了 Hadoop 的开发过程，并介绍了如何在 SCV 用例中使用 Hadoop。最后，有两块内容分别介绍了适合使用 Hadoop 的场景以及不适用的场景。按照惯例，还介绍了一些可以代替 Hadoop 的技术选型。

阅读完本章后，读者将对数据存储层以及 Hadoop 技术有非常清晰的认识。本章讨论的范围并没有包括 Hadoop 技术的全部，因此仅仅简要讨论了 Hadoop 的核心方面，这部分内容是实现本书用例的关键。

Hadoop 是实施数据湖的核心技术之一，希望通过阅读本章读者能够掌握这项技术。

恭喜读者，到现在为止已经完整掌握数据湖技术栈了。

仅供非商业用途或交流学习使用



Chapter 10 第 10 章

使用 Elasticsearch 存储全文索引

在前一章中对 Hadoop 进行了介绍，我们将数据存储在 Hadoop（HDFS）上。从 Hadoop 快速读取或查询数据是一个问题，此时需要能够存储全文索引数据的 Elasticsearch，它对数据湖的实现有着重要的意义和价值。

正如本书的其他章节，从这一章开始，将解释本章技术会在数据湖的哪个层级中被使用。然后解释选择这种技术的原因，之后开始深入研究 Elasticsearch 和它的工作原理。我们将覆盖 Elasticsearch 尽可能多的细节，以保证读者足以了解这项技术。虽然本章会提供尽可能多的细节，但是对 Elasticsearch 全面而深入的研究超出了本书的范围。

然后，将带领读者亲自编写一个代码示例，首先学习安装 Elasticsearch，然后在实践案例中操作它。最后让读者领会到 Elasticsearch 与 SCV 用例及数据湖实践的关联。

最后，将了解到哪些场景适合使用全文索引，哪些场合不适合使用全文索引，以及其他可用于替代 Elasticsearch 的技术。

现在，开始我们的探索之旅吧！

10.1 数据湖背景中的数据存储层与 Lambda 快速处理层

数据存储层是存储数据的地方。我们的核心持久化存储系统是 HDFS，但是由于它在查询方面性能较差，需要在其之上利用某种技术提供快速查询功能。对数据进行索引之后，可创建适当的快速视图并将其保存在 Lambda 快速处理层中。Lambda 快速层的职责是对 HDFS 中的数据进行索引，并对外提供高性能、可扩展的查询能力。

这里选择 Elasticsearch 作为快速层的技术框架，它能有效地满足 Lambda 快速处理层的

仅供非商业用途或交流学习使用

要求。由于 Elasticsearch 的突出特点，不用作过多的考虑就选择了它。

下一节将详细讨论 Elasticsearch，以便读者能清楚地了解该技术，并详细了解数据存储层。

10.1.1　数据存储层与 Lambda 快速处理层

在第 2 章，读者已经了解到数据存储层和 Lambda 快速层。数据存储层的职责是将收集到的数据持久化保存到数据湖中。Lambda 快速层的职责是对存储在数据存储层中的数据进行索引，并创建适当的快速视图。图 10-1 会刷新读者的认识，以图形化方式更好地向读者介绍快速处理层。

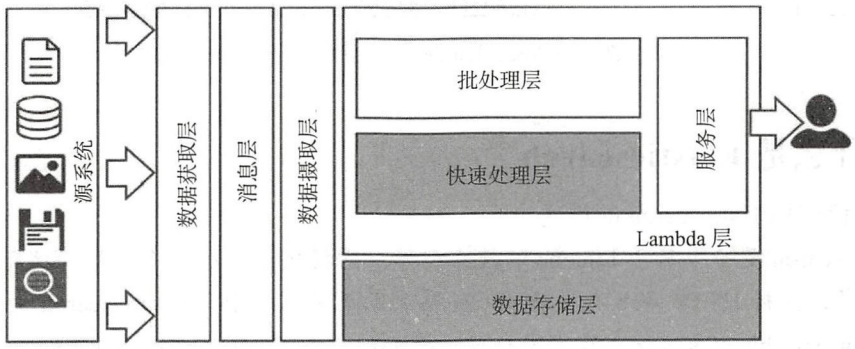


图 10-1　数据湖——数据存储层与 Lambda 快速处理层

在前一章中已经对数据存储层做了详细的介绍，因此这里不再赘述。Lambda 快速处理层基于数据存储层中的数据，构建了多种快速视图。数据存储层已经在前一章中介绍过了，相信读者现在仍记忆犹新。

10.1.2　数据存储层与 Lambda 快速处理层——技术路线图

本章的目标是覆盖全文索引数据的存储机制。本章将深入研究 Elasticsearch 那些令我们倾心的优良特性。

图 10-2 中用具体技术填充了概念架构的对应层级，这种方法将贯穿全书始终。后面会介绍每一种技术，以及其与数据湖整体架构的关联，在本书的第三部分，会将所有的技术整合到一起。

对于我们的 SCV 用例，已经从各种数据源系统收集了数据，并持久化保存在 HDFS 中。Elasticsearch 是构建在 HDFS 上的一层，它可以进行数据索引，并确保提供快速查询以满足用户的查询需求。

下一节将提供更多关于 Elasticsearch 的详细信息，并将使读者熟悉该技术的多种特性及工作原理。

仅供非商业用途或交流学习使用

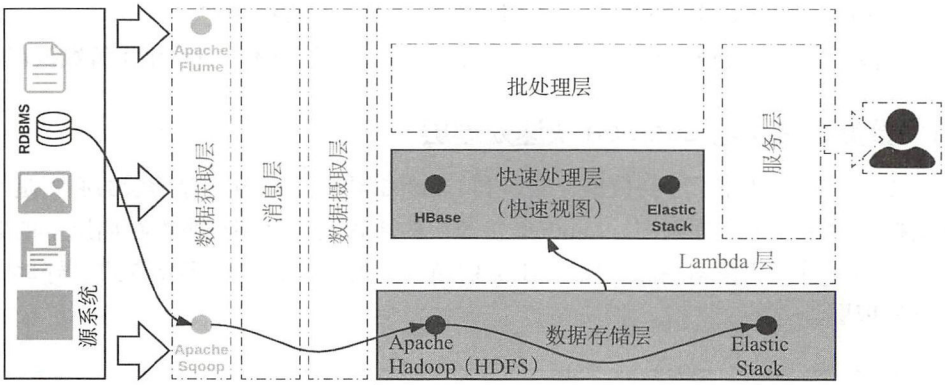


图 10-2 数据存储层、Lambda 快速处理层的技术路线图

10.2 什么是 Elasticsearch

维基百科对 Elasticsearch 的描述如下：

Elasticsearch 是一个基于 Lucene 的搜索引擎。它提供了一个分布式的、功能齐全的全文搜索引擎，支持 HTTP Web 界面及 JSON 格式的无模式文档。Elasticsearch 由 Java 语言开发，遵循 Apache 开源协议。官方客户端支持 Java、.NET（C#）、Python、Groovy 和许多其他语言调用。

前面的描述涵盖了 Elasticsearch 的各个方面，并详细描述了它的工作原理和核心功能。它是分布式的、可扩展的、企业级的，提供了高性能的查询，并且它是通过对 HDFS 中数据进行索引来实现这一点的。它也有丰富的 API，并且支持多种编程语言。本节会着力把较高层面的问题描述清楚，因为本章的后面相当一部分将用来解释 Elasticsearch 和机器工作原理。

10.3 为什么使用 Elasticsearch

对于我们正在寻求的功能，Elasticsearch 的技术是最先进的，这也是选择它的主要原因。数据湖的实现需要某些技术能力，以下列举了选择 Elasticsearch 的关键原因：

- 与 Hadoop 兼容（数据湖的持久化存储手段）。
- 分布式。
- 可伸缩性。
- 索引数据的能力。
- 高性能（快速查询 / 搜索）。
- 久经考验的技术（具有企业所需的所有能力）。
- 能处理海量异构数据。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

- 故障恢复与数据冗余能力。

10.3.1　Elasticsearch 简史

与大数据领域的其他技术相比，Elasticsearch 的历史很短。这并不意味着这项技术不成熟，相反，它是一个成熟的（70 多万次下载）的，并且有一个有活力的社区（7 万多社区成员）支持的技术。

2015 年 3 月，Elasticsearch 业务更名为 Elastic。

下面以图形的方式简要回顾一下 Elasticsearch 的发展史，与其他技术框架的介绍方式类似，如图 10-3 所示。

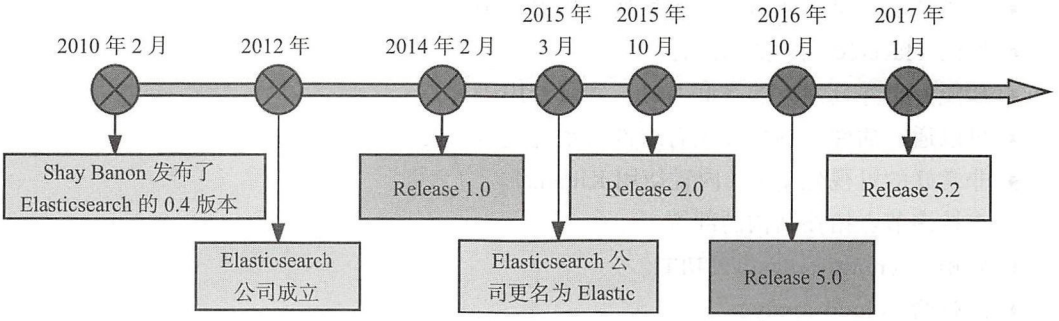


图 10-3　Elasticsearch 简史

Elasticsearch 与业界的一些其他产品紧密捆绑在一起，图 10-4 显示了 Elasticsearch 及相关产品一起演变的过程。知道这一点对于了解 Elasticsearch 的历史是相当重要的。

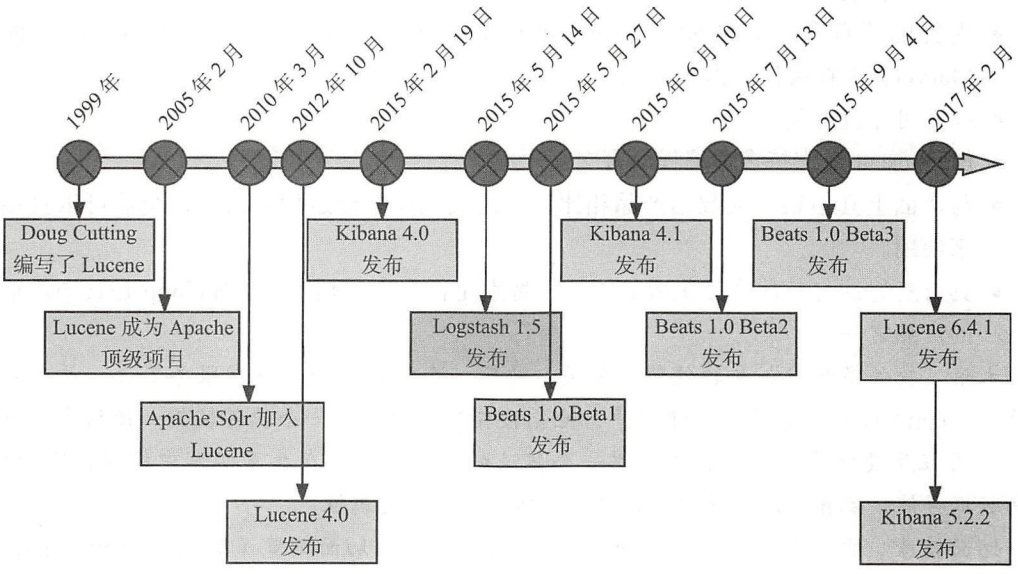


图 10-4　其他与 Elasticsearch 共同发展的产品

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

前面的版本数字有点混乱，但显示了各种 Elasticsearch 辅助产品及其发展历史方面的知识。

10.3.2 Elasticsearch 的优势

以下是 Elasticsearch 的一些优点：

- 开放源码（遵循 Apache 2 协议）和强大的社区支持。
- 天然支持分布式。
- 天然支持 RESTful（通过 HTTP 传输 JSON）。
- 操作非常快。
- 搜索功能齐全：能够支持各种搜索相关的用例。
- 切面（faceted）搜索 / 分析。
- 功能齐全并且强大的查询 DSL（领域专用语言）。
- 可以通过插件机制集成现有组件对系统进行扩展。
- 非常好的可视化支持（内置使用 Kibana）。
- 支持多节点拓扑结构的部署。
- 多租户（multi-tenancy）功能。
- 云就绪（cloud-ready）。
- 对开发人员友好，使用熟悉的语法。
- 内置高效缓存机制，有利于快速查询。
- 高度可伸缩：单节点到数百个节点，支持水平扩展，可按需处理 PB 量级的数据。
- 高可用，使用灵活。
- 内置的管理和监控功能，能提供足够多的细节来帮助维护企业产品（以前称为 Marvel，现在被打包为 X-Pack）。
- 高度可靠且可预测。
- Elasticsearch 支持多种编程语言交互。
- 与市面上其他很多大数据产品相比，它的安全策略经过精细设计，企业可以直接拿来使用。
- 具有良好的与其他产品集成的能力，特别是很多大数据产品，Hadoop 是其中最重要的产品之一。

术语“软件多租户”与软件架构相关，此时一个软件实例运行在服务器上，并为多个租户（tenant）服务。租户是一组用户，他们按指定权限共享一个软件实例。使用多租户架构，应用程序设计为每个承租者提供一个候选的共享实例，在共享实例中提供以下功能：包括数据存储、配置、用户管理、租户个体功能和非功能属性。

切面搜索，也称为切面导航（faceted navigation）或切面浏览（faceted browsing），是一种根据切面分类系统来访问信息的技术，允许用户通过使用多个过滤器来探索信息集合。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

切面分类系统将每个信息元素按照多个显式的维度进行分类，称为 facet，使分类可以按多种方式来访问和排序，而不是单个的、预先确定的分类顺序。

——维基百科

10.3.3　Elasticsearch 的劣势

Elasticsearch 有其核心设计原则，因此对它的使用应该扬长避短。实践中，它通常被用于某些其不太擅长的场景中。当在某些用例中使用时，它会产生事与愿违的结果，这些方面可视为 Elasticsearch 的劣势：

- 不应用作主要的持久化存储手段。因此，除了 Elasticsearch 之外，还需要另一个存储系统来实现真正的持久化。
- Elasticsearch 无法满足开箱即可用的 ACID 要求。
- 尽管生产环境中的 Hadoop 不是在廉价硬件（commodity hardware）上运行，但理论上它确实支持这种做法。在生产状态中，Elasticsearch 本质上不会运行在廉价硬件上。
- Elasticsearch 有垃圾收集（Garbage Collection，GC）方面的问题。
- 产品更新有时会很麻烦。在生产环境中，索引重建、数据合并等可能会比较烦琐。
- 为了在生产环境中有效地进行管理，Elasticsearch 通常被认为学习曲线很陡峭。学习曲线陡峭是非常主观的，但使用者不能回避这一点。
- 对需要进行事务处理的用例的支持有限。
- 对海量数据集进行扫描式搜索或高级计算时，Elasticsearch 并不是很好的选择。

10.4　Elasticsearch 的工作原理

在本节中将更详细地讨论 Elasticsearch，包括它的体系架构、主要组件、各种术语及重要概念。这是本章最有趣的部分，目的是让读者理解这项技术，然后在实际的编程练习中操练 Elasticsearch。

10.4.1　Elasticsearch 架构的核心原则

Elasticsearch 的核心架构原则如下所述：

- Elasticsearch 很大程度上是由内存驱动的，索引和数据服务在内存中进行，数据也会持久化存储在存储设备上。
- Elasticsearch 构建于 Apache Lucene 引擎之上，Lucene 提供了核心的索引功能，而索引分布、可伸缩性和其他特性则由 Elasticsearch 来管理。
- Elasticsearch 在设计之初就考虑了分布式可伸缩，它的核心是水平扩展架构。
- 从本质上来说，Elasticsearch 是一种更可靠的系统，它是弱一致性系统，能保障最终

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

一致性。

- 支持对索引和数据进行备份，这些索引和数据可以动态修改，以保证读取和写入的可用性。
- 由于其高度分布式的架构，Elasticsearch 可以很容易为结构化或非结构化数据存储扩展到 PB 级别。
- 对分析器（analyzer）或分词器（tokenizer）的使用必须谨慎，考虑到搜索效率，需要利用分析器或分词器产生最佳数量的词条（token）。
- 对索引进行分区，这样索引是基于关键属性分区的，可以独立地删除。
- 大型集群（节点数 >10）部署时需要为 master 节点单独部署一台服务器，而小集群 master 节点所在服务器可以客串数据处理节点。
- 所有的 Elastic 节点的底层架构的配置必须一致，以便于实现负载平衡。
- 所有的 Elastic 节点都必须在同一个网络中，这样就可以精确地确定每对节点间的距离，以避免脑裂（split brain）。
- 单个集群可能无法跨越数据中心。只有当端对端网络延迟小于 10 毫秒时，才有可能实现交叉数据中心集群。跨数据中心的部署也可能限制集群的可扩展性。

10.4.2 Elasticsearch 术语

Elasticsearch 的架构很容易理解，在了解其基本工作原理之前，理解其相关术语是很重要的。本节内容的目的是解释 Elasticsearch 的相关术语。要把这些术语解释清楚是很困难的，因为很多术语是等价的、可互换的。不过读者大可放心并继续阅读，在术语介绍相关小节的最后部分，读者将对所有术语有清楚的理解，这对于理解本章的其余部分至关重要。

1. Elasticsearch 中的文档

Elasticsearch 中的数据以文档的形式存储。这些文档是可寻址的，并且可以在已标识过的字段中进行搜索。样本文档（JSON 格式）如图 10-5 所示。

```
{
    "first_name": "John",
    "last_name": "Doe",
    "dob": "01-01-2017,
    "communication": {
        "email": "xx@xx.com",
        "mobile": "0097112345678"
    },
    "other_attributes":{
        "field1": "field_value",
        "field2": "field_value"
    }
}
```

图 10-5 Elasticsearch 文档范例

与 Java 中的字符串对象相似，Elasticsearch 中的文档是不可变的，所以一旦创建，就不能更新它们。如果想要更新一个文档，必须重新索引或者替换它。

除了实际的文档数据之外，文档还有关于文档本身的附加信息，称为文档元数据。文档中最重要的元数据如下：

- _index：Elasticsearch 中文档的逻辑分组，相关的文档被放在同一个索引中。例如，

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

一个组织中的所有客户索引都可以保存在 customer 索引中。_index 的值不能以下划线开头，也不能包含逗号，并且应该小写。

- _type：文档对象所属的类型。这允许 Elasticsearch 能对同一个索引中的数据逻辑进行进一步分割。Elasticsearch 中的索引通常存在子分类。_type 的值不能以下划线或句点开头，但可以是小写或大写字母；与 _index 类似，它们不能包含逗号，并且限制长度为 256 字符。
- _id：这个字符串（_id）通过组合 _index 及 _type 来唯一标识一个文档。可显式提供也可自动生成这个值。

在面向对象和函数式编程语言中，不可变对象（immutable object）是一个对象，它的状态在创建之后不能被修改。

——维基百科

图 10-6 中展示了 Elasticsearch 中的多个文档。

2. Elasticsearch 中的索引

Elasticsearch 中存储在一个逻辑分组中的数据被称为索引。在同一个索引中，存储的文档具有相似的特性。索引的定义与设置对 Elasticsearch 的使用有极大的影响，会影响搜索、更新等诸多方面。在集群中，可以根据业务需要定义任意数量的索引，但是需要对 Elasticsearch 集群及其节点进行妥善配置。单个索引可以存储大量的数据（数十亿的文件），容量可高达 1TB。不过在一个索引中存储这么多数据可能会影响搜索或选择数据时的性能。创建索引时，可以指定需要的分片和副本的数目。

ID	文档
1	data lake for enterprises
2	data enterprises for lake
3	enterprises lake
4	data lake
5	lake data
6	enterprises

图 10-6　Elasticsearch 中的多个文档（字符串格式）

图 10-7 中展示了倒排索引示例。

3. Elasticsearch 中的分片

在 Elasticsearch 中，数据被存储及索引在分片（shard）中，索引逻辑上由一个或多个分片构成，同一个索引中的数据描述了同一类业务对象。分片更像是 Elasticsearch 的内部机制，而索引是通常人们理解的 Elasticsearch 的数据单元。分片是 Elasticsearch 中的独立单位，可以在 Elasticsearch 集群中的多个节点之间分发。切片机制是 Elasticsearch 可水平扩展能力的基础。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

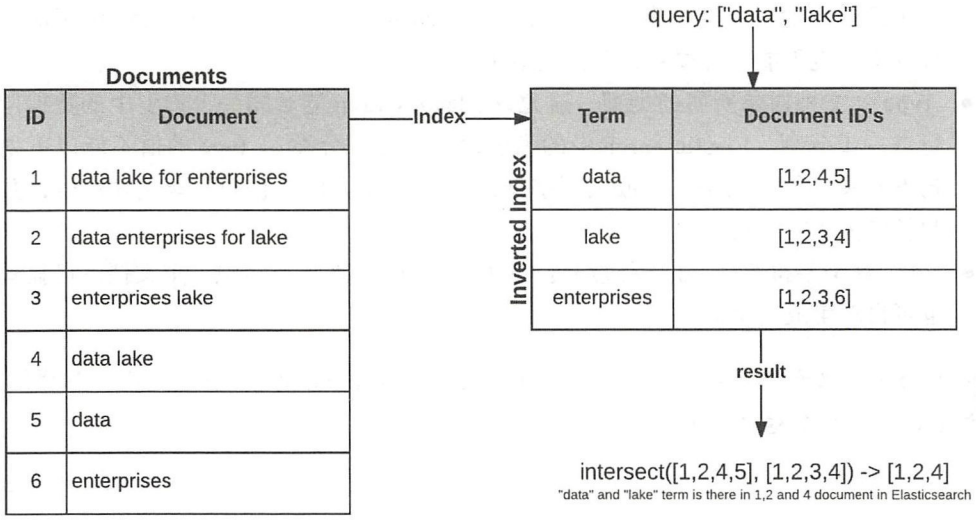


图 10-7 文档与索引（倒排索引）

可以根据需要将索引复制到一个或多个分片中。一旦被复制，索引就会包含以下分片：

- 主分片（primary shard）：用于创建副本的分片。
- 副本（replicated shard）：从主分片复制出来的分片。

一个索引可以定义在创建索引时需要的分片数和副本数。

4. Elasticsearch 中的节点

Elasticsearch 节点（node）即集群中的一个 server，集群可以由一个或多个节点（server）构成。Elasticsearch 集群中的节点标识是唯一的，集群启动时默认会为每个节点分配 UUID，如果需要，也可以为每个节点指定唯一的名称。节点名称非常重要，可以用于各种各样的管理目的。节点默认加入名为 Elasticsearch 的集群，但是也可以根据需要配置为加入指定名称的集群。即使集群中只有一个节点，在启动时，Elasticsearch 会构建一个只有这个节点的集群。

通用唯一识别码（UUID）是一个 128 位数字，用于识别计算机系统中的信息。当根据标准方法生成时，UUID 追求的是实践意义上的唯一，不需要通过中央控制端来验证唯一性，也不需要生成 UUID 的各方互相协商，这与大多数其他编号方案不同。虽然一个 UUID 重复的概率不是零，但是它非常接近于零，几乎可以忽略不计。

——维基百科

5. Elasticsearch 中的集群

前面已经详细介绍过了，集群由一个或多个节点（server）构成，集群作为一个 Elasticsearch 实例对外提供查询服务。图 10-8 中描绘了工作场景中的集群、节点和分片（主分片及副本）。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

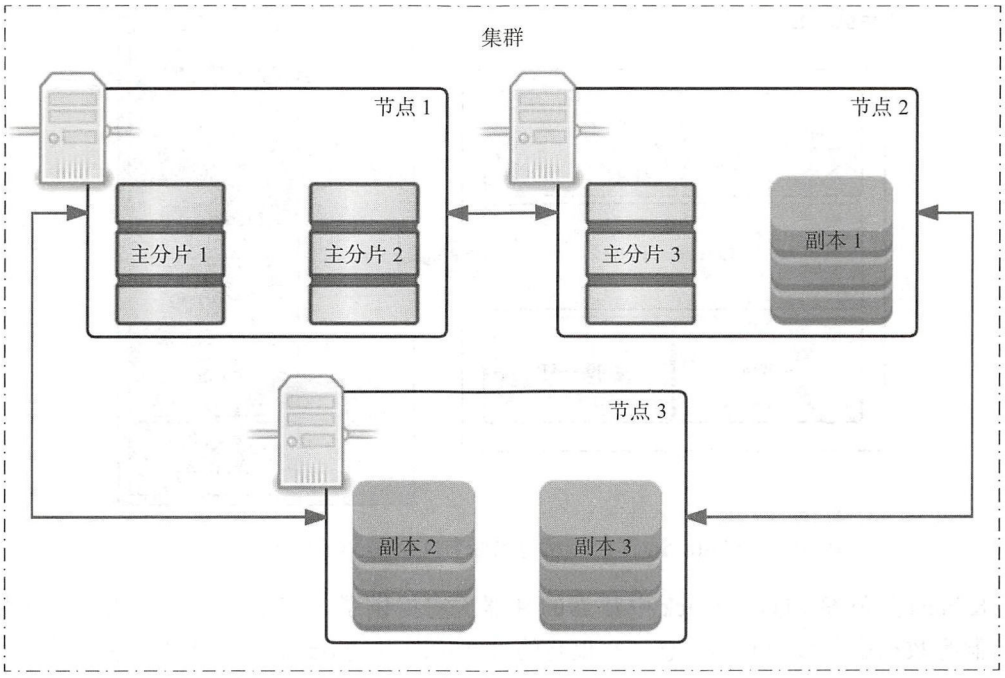


图 10-8　包含了节点和分片（主分片及副本）的典型的 Elasticsearch

10.5　Elastic Stack

使用 Elasticsearch 1.x 和 Elasticsearch 2.x，各种产品之间协同工作时，必须单独选择和处理。这依赖先前的多方面的 Elastic Stack 经验。Elastic Stack（5.x）出现之后，将 Elasticsearch、Logstash、Kibana、Beats 和 X-Pack 打包在一起，真正形成了一个技术栈（平台），在不同版本上的尝试的工作量及出错概率都很小。与早期版本相比，使用 Elasticsearch 变得更加容易了。

图 10-9 显示了 Elastic Stack 的所有组件是如何协调工作的。

在下一小节中，我们将讨论 Elastic Stack 的每一个组件，确保读者了解足够多的技术细节来理解 Elastic Stack。同样，也不会对每个组件做过于艰深的探究，但是读者运用了解到的知识足以实现手头的范例。

10.5.1　Kibana

Kibana 是了解 Elastic Stack 的窗口。

Kibana 为用户提供了数据可视化，它也是一个可扩展的用户界面，用于配置和管理 Elastic Stack 的所有方面。

——elastic.co/v5

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

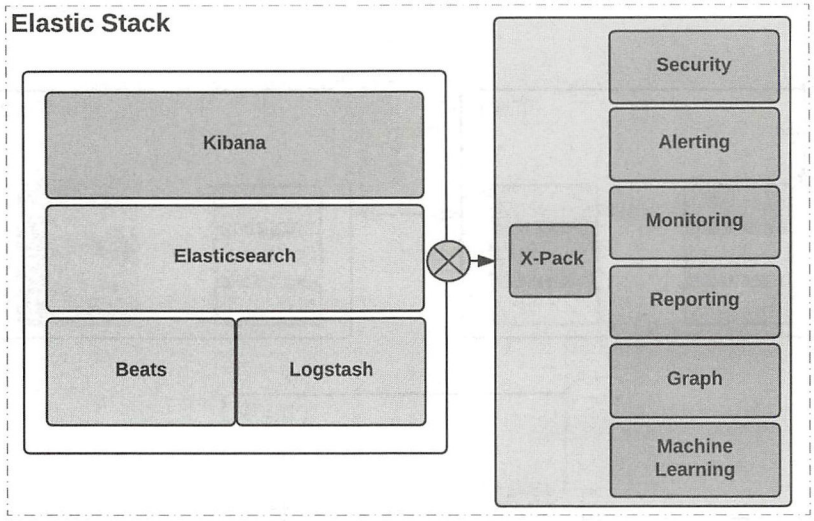


图 10-9 Elastic Stack 5.x（所有图标来自 https://www.elastic.co/v5）

Kibana 是洞察 Elastic Stack 中数据的利器。它提供了丰富的可视化能力，用于帮助用户洞悉数据中蕴藏的有用信息。它提高的可视化图表包括经典的线形图、柱状图、直方图以及其他复杂的图形，如时间序列图、图谱数据可视化、地理位置信息（地图）的可视化等。

图 10-10 和图 10-11 展示了 Kibana 仪表盘（dashboard）子图和仪表盘，其中有经典的饼图和柱状图。

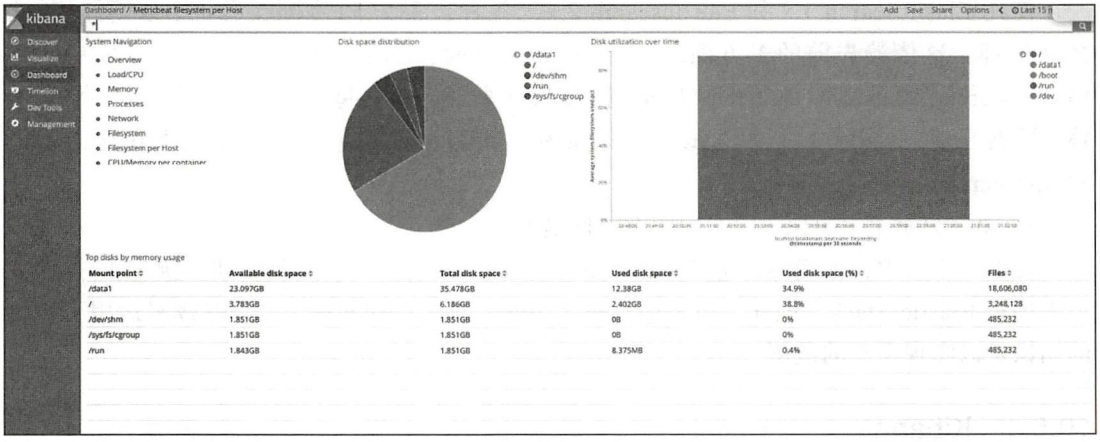


图 10-10 Kibana 仪表盘子图

与图 10-10 类似，图 10-11 显示了 Kibana 仪表盘，它有更多的图形来增强可视化能力。

Kibana 还通过帮助管理员更有效地管理和监控 Elastic Stack，提供了 Elastic Stack 的可视化表示。在 Kibana 的管理页上有一个这样的例子，如图 10-12 所示。

非卖品，仅供非商业用途或交流学习使用

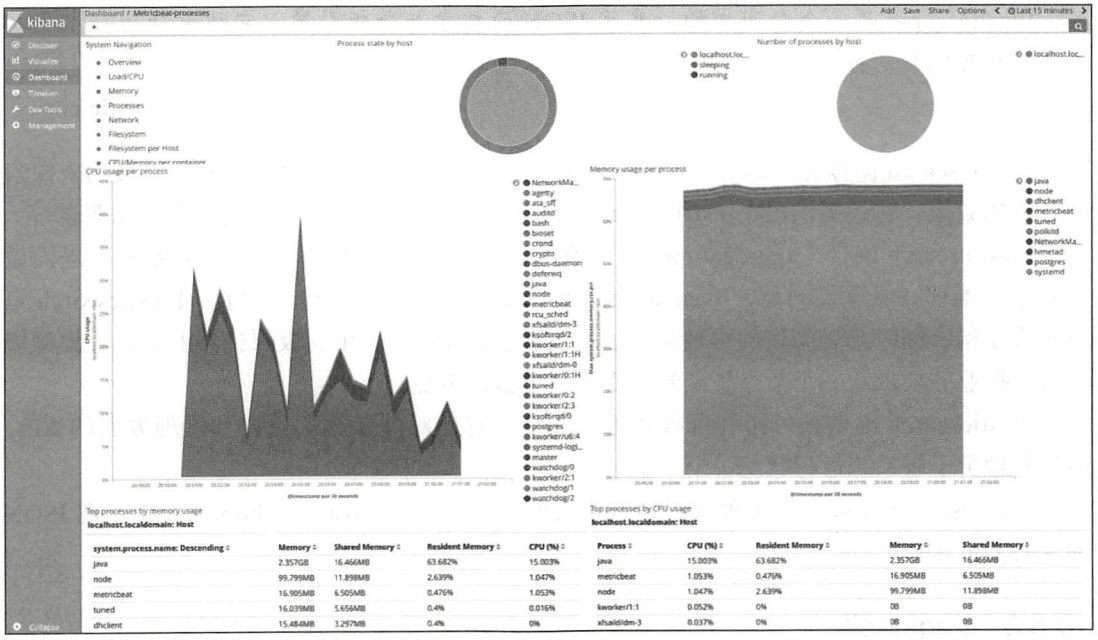


图 10-11　Kibana 仪表盘

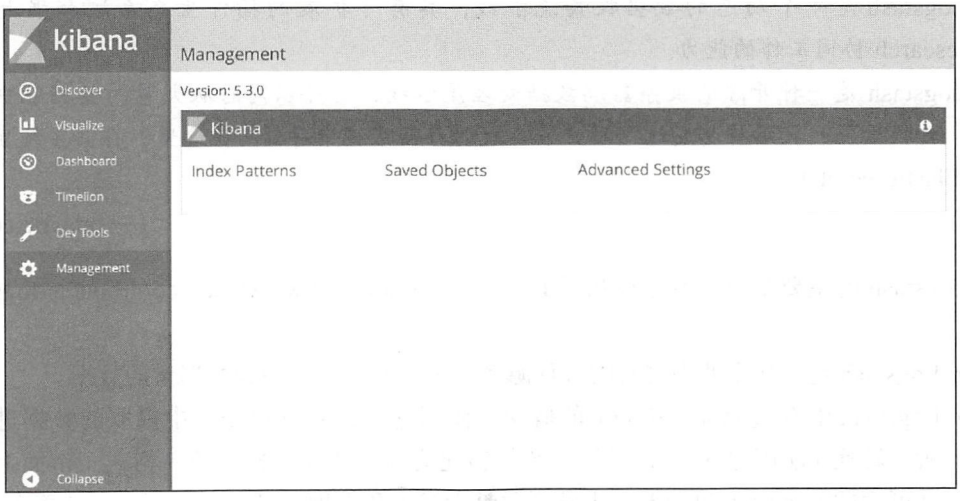


图 10-12　Kibana 中的 Elastic Stack 管理页

Kibana 还提供了开发者工具（例如控制台），可以使用它来查询 Elastic 中的数据。它还提供了自动完成（auto-completion）等功能，这对开发人员非常有用。

10.5.2　Elasticsearch

Elasticsearch 可用于搜索、分析、存储数据，它是 Elastic Stack 的核心。

Elasticsearch 是一个分布式的，基于 JSON 的搜索与分析引擎，它被设计为可水平扩展，高度可依赖，易于管理。

——elastic.co

Elasticsearch 被称为 Elastic Stack 的核心，它被很多企业用于存储那些可以被用来探索和分析的数据。它还提供很多了 RESTful API，使用这些 API 可对数据进行索引和搜索。

Elasticsearch 聚合（aggregation）方法有助于搜索来自源系统的海量日志数据，进而做出有用的推理。最重要的是，Elasticsearch 提供了高性能的全文检索功能。Elasticsearch 是分布式架构，同时具有高度的可伸缩性，并且能在廉价 PC 或企业级服务器上运行，集群规模可根据业务要求从一个节点扩展到数百个或更多的节点。

Elasticsearch 也是高可用和可容错的。此外，它还对外暴露了监控和管理方面的 API，便于管理员了解系统运行状况。

Elasticsearch 还提供了多种编程语言实现的客户端，如 Java、Python 等，能处理 JSON 和 RESTful API。

10.5.3 Logstash

Logstash 用于聚集、转换和存储数据。Elastic Stack 中的 Logstash 如图 10-13 所示。

Logstash 是一个动态的数据收集流水线，具有可扩展的插件生态系统和强大的与 Elasticsearch 协同工作的能力。

Logstash 是一种开源的服务器端数据处理流水线，它可以同时从大量的数据源中摄取数据、转换数据，然后将其发送到用户最喜欢的存储系统中存储（在这里，存储系统显然指的是 Elasticsearch）。

——elastic.co

Logstash 的主要特性（参考最初的 Elastic 网站 https://www.elastic.co/products/logstash）如下：

- Logstash 允许从企业中现有的各种源系统中摄取不同类型和数据量的数据。
- Logstash 中有过滤器（filter）的概念，使用它可以对从数据流中摄取的数据进行解析、转换（使用过滤器），最终将数据转化为易于快速分析的统一格式。市面上有多种可用的 Logstash 过滤器，由开源贡献者维护和管理，可以在下面这个链接中找到：https://www.elastic.co/guide/en/logstash/current/filter-plugins.html。
- 一旦数据被摄取并转换，Logstash 可以让我们将输出结果存储到各种各样的存储系统中去（在我们的例子中，是将索引数据存储在 Elasticsearch 中）。支持的存储系统的完整列表可以在这里找到：https://www.elastic.co/guide/en/logstash/current/output-plugins.html。
- Logstash 设计时就考虑了可扩展性，这可以通过编写插件来实现。要审慎地使用这

些插件，它们可构建和处理任意所需的数据处理流水线。现在已经有了一个丰富的插件生态系统，而且它在开源贡献者的支持下也不断地增长。

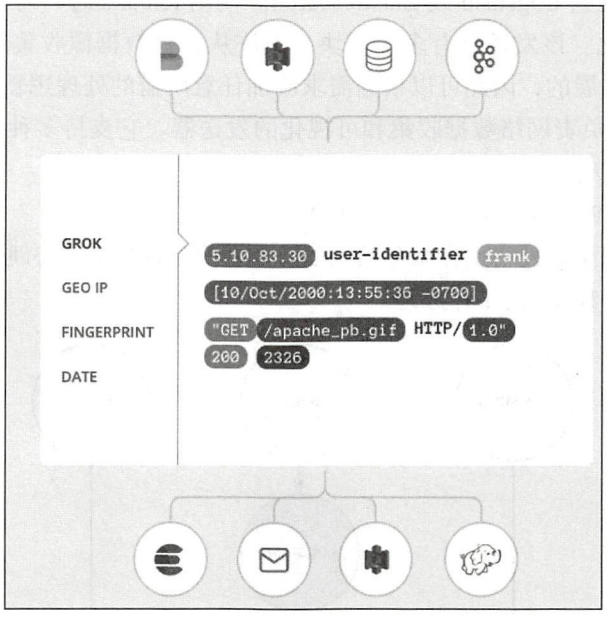


图 10-13　Elastic Stack 中的 Logstach（https://www.elastic.co/products/logstash）

10.5.4　Beats

Beats 是一个轻量级的数据发送平台，可以将数据从边缘机器（客户端）发送到 Logstash 和 Elasticsearch。

Beats 是一个单一用途的数据发送平台。它们作为轻量级代理被安装，并从成百上千的机器发送数据到 Logstash 或 Elasticsearch。

——elastic.co

Beats 运行在服务器上，根据数据类型，它采用所谓的“发送器”（shipper）来收集数据。这些发送器收集的数据可以直接发送到 Elasticsearch 中，或者可以根据需要将数据发送给 Logstash，经过适当的转换以后发送到 Elasticsearch（实际上可以是前一小节中详细介绍过的任何输出存储系统）。

在这里列出了一些 Beats 内置的发送器，如果需要，它们也可以被扩展。有很多可用的发送器，由开源贡献者维护和管理。完整的列表请参考 https://www.elastic.co/guide/en/beats/libbeat/current/community-beats.html。

- Filebeat：用于收集服务器中生成的日志的轻量级发送器。它通过将日志行记录转发到 Logstash 或 Elasticsearch 来实现这一点。

- Metricbeat：用于收集各种度量指标的轻量级发送器。Metricbeat 用于收集服务器（内存、CPU 等）的各种度量指标，以及其他软件运行指标（如 Nginx、Redis 等），并将这些数据转发到 Logstash 或 Elasticsearch。使用 Kibana 时，可以用适当的方式将这些指标可视化。该发送器有多个模块，允许从各种数据源收集这些度量指标，这些模块也是可扩展的，因此可以根据需求添加任意所需的处理逻辑。
- Packetbeat：负责网络数据收集和可视化的发送器。它支持多种协议，并且允许用户编写插件来进行扩展。
- Winlogbeat：Windows 事件日志发送器。
- Heartbeat：用于监控宿主在服务器上的应用程序实时状态（心跳监控）的发送器。

图 10-14 显示了 Beats 的工作原理。

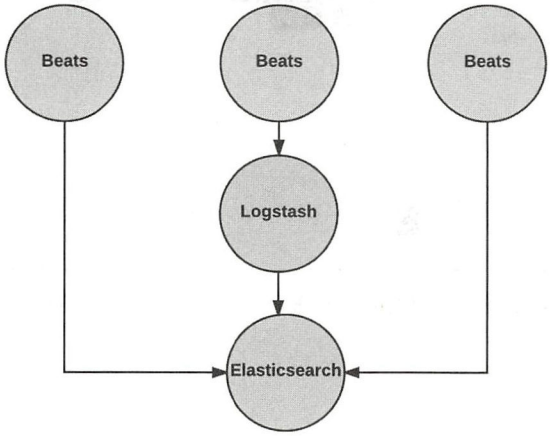


图 10-14 Beats 的工作原理

X-Pack

X-Pack 是由 Elastic 的工程师开发维护的 Elasticsearch 的一个扩展包，它集成了多种有趣的特性，用户可以放心地在 Elastic Stack 中使用它。

——elastic.co

这个最重要的组件横跨了了我们之前讨论过的 Elastic Stack 中的所有组件。在 Elastic 5.x 之前，这些组件都是分开的，为了得到一个功能齐全的企业级 Elasticsearch 实例，必须把它们整合在一起。有了 Elastic Stack 之后，情况大为改观，此时有了一个功能完备的、完全集成的套件，所有这些组件无缝协作。这些特性由 Elastic 的工程师创建及维护，可确保 Elastic Stack 正常升级时组件之间不会互相冲突。

Elastic Stack 的大部分非功能方面都由 X-Pack 组件负责，下面列举了最核心的方面：

- Security：可以用于支持集群安全。它还具有内置的授权机制，基于角色和权限的形式来轻松进行系统管理。也能够与 Active Directory(AD）和轻量级目录访问协议（LDAP）

集成，它还能够与组织内的自定义身份和访问管理（IDAM）系统进行集成，甚至可以通过 X-Pack 实现数据级别的安全策略。它还支持通过在节点之间启用 SSL 通信，以及根据特定的使用需求保护和屏蔽数据的能力来保护基础设施。X-Pack 的安全特性还包含对各种系统和用户活动的审计，请记住，这只是其审计功能的一部分。

- Alerting：帮助配置各种警报。一个例子是通过观察各种与系统健康相关的指标来触发警报。它有助于查看各种系统级参数，并根据需要触发已配置的警报。它也具备感知 Elasticsearch 中数据变化的能力，可以触发已配置警报。很多现有的 X-Pack 整合方案中，预警机制是可以选择的。如内置的 E-Mail，又如 Slack、PagerDuty、HipChat 等预警机制。它还具有 WebHook 的概念，任意 URL 均可以配置，当特定规则满足时，可调用该 URL，这可以作为一种报警机制。
- Monitoring：让用户了解了 Elastic Stack 是如何运行的。它监控集群、集群中的节点、Elasticsearch 中的各种索引、Kibana 和 Logstash，并对监控数据进行可视化展示，以便进行分析。它还会在必要的时候存储收集到的各种数据，这些数据可用对现状及历史情况进行分析。
- Reporting：允许创建各种报告。这些报告可以根据不同的规则进行调度和触发，并使用不同的机制以不同的格式来发送。
- Graph：帮助分析存储在 Elasticsearch 中的数据的关系。它提供了一种看待数据的新方法，可以提供对现有数据深入而新颖的洞察力。

Web 开发中的 WebHook 是一种通过自定义回调函数来扩展或改变 Web 页面或 Web 应用程序行为的方法。这些回调可以由第三方用户和开发人员维护、修改和管理，这些用户可能不一定与原始网站或应用程序有关联。

——维基百科

10.6　Elastic Cloud

Elastic 官方将 Elasticsearch 视为一个对企业来说开箱即可用的服务。Elastic Stack 可扩展性强，并且一直在升级。更多细节请参考 https://www.elastic.co/cloud/enterprise。

Apache Lucene

Elastic Stack 中的 Elasticsearch 组件基于 Apache Lucene。Elasticsearch 底层使用了倒排索引（Inverted Index），它是 Apache Lucene 对文档进行索引的基础。

Apache Lucene 是一个高性能的、功能完备的文本搜索引擎库，完全由 Java 语言开发。它适用于几乎所有需要全文检索的应用程序，尤其是跨平台的应用。

——Apache Lucene

Doug Cutting 是 Lucene 的创始人，同时也是 Hadoop 的创始人。Apache Lucene 是一个高性能的搜索引擎库，能够提供高级的搜索选项，它使用了倒排索引，构建索引时，文档中的词项被提取，并作为关键字被存储在索引中。索引中也存储了与每个词项相关的文档 ID 集合（Elasticsearch 中以文档为存储单位）。一个文档（document）可能出现在多个索引中，一个索引中也可能会包含多个文档。每个文档可以包含多个字段（field），每个字段可以拥有多个值（词项）。

以下是 Elasticsearch 的一些重要特性：

- 在 Lucene 内核之上提供一个方便的抽象层，这使得使用 Elasticsearch 更加便捷容易。
- 基于分片，每个分片是一个独立的 Lucene 实例。
- 在 Lucene 之上构建了分布式计算能力，提供了多个 API 与 Lucene 交互。
- Lucene 之外还提供了很多其他的特性，如丰富的监控与管理功能。

Lucene 工作原理

Lucene 所扮演的主要角色是在应用程序中支持搜索需求。虽然应用程序可能会实现端到端用例（end-to-end use case），但搜索可能需要一些非常专业的功能，这可以由 Lucene 提供。

Lucene 的基本操作单元为文档。信息由某种外部机制进行捕获，以文档方式存储在 Lucene 中。将信息 / 内容以文档形式保存的主要原因是信息 / 内容需要保存在一种易于被 Lucene 引擎解释的格式中。

一旦创建并存储了文档，就会根据所提供的模式（schema）或映射（mapping）分析文档，Lucene 引擎决定要索引什么内容及不索引哪些内容。文档分析完成后，文档的特定字段会进入索引过程，包括使用预先确定的分词器（tokenizer）分词产生词条（token）。词条会被映射到倒排索引中对应的词项中，对应的文档 ID 也会被追加到索引[⊖]中，索引完毕后，便已经准备好了要搜索的词条了。

此外，分词器中还可以包含多个词条过滤器（token filter），这样可以对某些词条进行自定义处理。既可以开发自己的分词器，也可以串行使用多个过滤器进行链式处理。

Lucene 引擎还提供了一个查询接口，可用于对存储在索引中的文档执行查询。Lucene 提供了自己的查询语言，也称为 Lucene 查询语言。一旦执行查询，就会使用查询中的词条来匹配倒排索引中的词条，默认情况下返回的是命中文档的 ID 集合。如果索引时保存了源文档副本，也会返回源文档。Lucene 构建倒排索引的过程如图 10-15 所示。

⊖ 准确地说是倒排链，即 posting list。——译者注

仅供非商业用途或交流学习使用

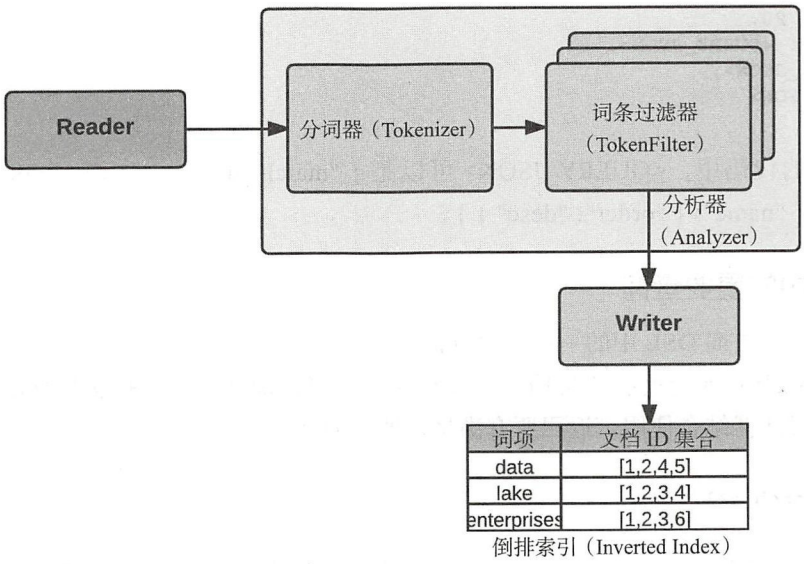


图 10-15　Lucene 构建倒排索引的过程

图 10-15 显示了在 Apache Lucene 背景中倒排索引的工作原理，图 10-16 将显示在 Elasticsearch 背景中的 Lucene 倒排索引。

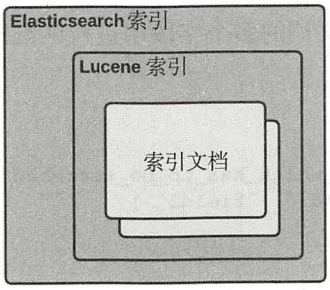


图 10-16　Elasticsearch 与 Lucene 索引的关系

10.7　Elasticsearch DSL

Elasticsearch DSL 是一种基于 Elasticsearch 的库，可以用它来编写和运行针对已存储数据的查询。它是对 Lucene 的查询语法的包装，使得这些交互变得更简单，允许使用 JSON 语法编写查询。

下面显示了 Elasticsearch 中的数据示例查询：

```
Curl -XPOST 'http://localhost:9200/customer/_search?pretty=true  -H
'Content-Type: application/json' -d '
{"from": 0,
```

仅供非商业用途或交流学习使用

```
"Size": 20,
"query": <QUERY_JSON>,
<FILTER_JSON>,
<SORT_JSON>
}'
```

在前面的代码中，<QUERY_JSON> 可以是 { "match_all": {} }，而 <SORT_JSON> 可以是 "sort": { "name": { "order": "desc" } }。

查询 DSL 中的重要查询

下面显示了查询 DSL 中的一些重要查询：

- match_all：匹配所有的文档。当没有确定任何查询细节时，默认执行该查询。它可以与过滤器结合使用，返回所有满足过滤器的文档。

```
{
    "match_all": {}
}
```

- match：当需要在文档任意一个字段中进行全文或精准匹配时使用该查询。

```
{
    "match": {
        "field_name": "phrase_which_has_to_be_searched"
    }
}
```

- multi_match：当需要在文档的多个字段中进行匹配时可使用该查询，它像是同时执行了多个 match 查询。

```
{
    "multi_match": {
        "query": "phrase_which_has_to_be_searched",
        "fields": [ "field1", "field2" ]
    }
}
```

- range：用于在数值类或日期类字段执行特定区间的查询。

```
{
    "range": {
        "field_name": {
            "<gt or gte>":  5,
            "<lt or lte>":   50
        }
    }
}
```

- term：用于精准的词项（term）查询。

```
{
    "term": {
         "field_name": "search_text"
    }
}
```

仅供非商业用途或交流学习使用

- terms：与 term 查询类似，只不过是允许使用多个用于检索的词项。

```
{
     "terms": {
          "field_name": [ "search_text1", "search_text2",
"search_text3" ]
     }
}
```

10.8　Elasticsearch 中的节点

如前面所述，在 Elasticsearch 中，节点就是组成集群的服务器之一。集群中的节点可以配置为以下类型：

- Master 节点
- Data 节点
- Client 节点

10.8.1　Master 节点

集群中的任何节点都可以成为 Master 节点。在 Elasticsearch 中，将 elasticsearch.yml 文件的 node.master 属性值设置为 true 即可。一旦 Master 节点由集群自动产生，这个节点就会被赋予一些关键的职责，如下所述：

- 为集群中的节点分配分片。
- 创建或删除索引。
- 向集群中其他节点广播系统状态以及接收其他节点的确认信息。
- 集群中有节点进入或退出时执行必要的动作。
- 定期 Ping 所有节点，并周期性地等待其他节点 Ping 回。如果 Master 节点由于任何原因挂掉，那么另外某个符合 Master 节点条件的节点会作为 Master 节点。

10.8.2　Data 节点

集群中的任意节点都可以成为 Data 节点。对于大的集群，实际上会有一个专门的节点来承担 Master 节点的职责而不存储任何数据（将 node.data 属性配置为 false 就可以避免让节点同时成为 Data 节点和 Master 节点）。Data 节点的主要职责如下：

- 该节点存储分片形式的数据。
- 索引、搜索数据，聚合计算。

通过设置 node.master 属性为 false，可以确保节点成为集群中的 Data 节点。对于企业级设置，这是集群的常规设置方式，更加可靠和可预测。

仅供非商业用途或交流学习使用

10.8.3 Client 节点

任意非 Master 节点（此时 node.master 节点属性设置为 flase）或非 Data 节点（此时 node.data 属性值设置为 false）的节点被称为 Client 节点。Client 节点充当了负载平衡器的角色，将请求或查询路由到合适的节点上去处理。一般来说，集群中没有必要存在此类节点，因为 Data 节点可以完成它们的工作。然而，大集群中需要有若干个这样的节点，以便减少集群中重要节点的工作量。

图 10-17 描述了 Elasticsearch 集群中各种节点的作用及关系。

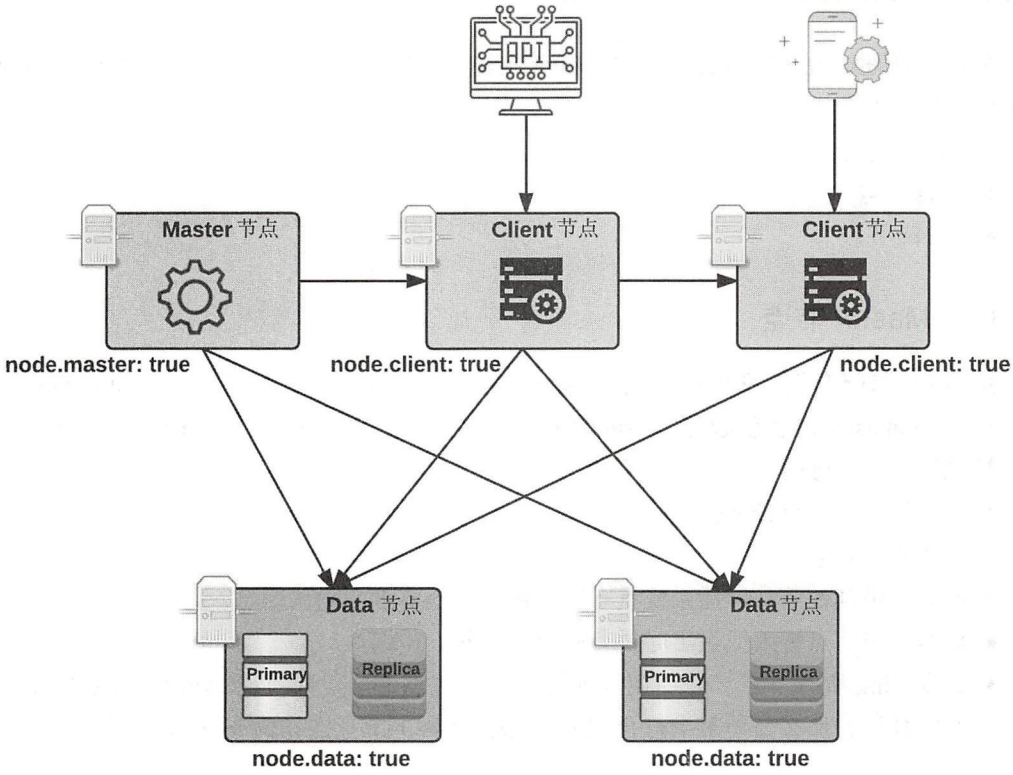


图 10-17 Elasticsearch 集群中的各种节点

10.9 Elasticsearch 与关系数据库

尽管已经对 Elasticsearch 术语做了详细的介绍，这里还需要借助在工业界占据主流地位的关系数据库中的概念对 Elasticsearch 术语做进一步的探究，两者存在很多相似之处。下面将对它们做一个比较，如表 10-1 所示。当读者使用任意 Elasticsearch 术语时，均可以联想到对应的关系数据库概念。

仅供非商业用途或交流学习使用

表 10-1　Elasticsearch、关系数据库术语对照表

Elasticsearch 术语	关系数据库术语
节点（Node）	数据库实例（DB Instance）
集群（Cluster）	数据库集群（DB Cluster）
索引（Index）	数据库（Database）
类型（Type）	表（Table）
映射（Mapping）	模式（Schema）
分片（Shard）	物理分区（Physical Partition）
路由（Route）	逻辑分区（Logical Partition）
文档（Document）	行（Row）
字段（Field）	列（Column）
Elasticsearch DSL（查询 DSL）	结构化查询语言（SQL）

10.10　Elasticsearch 生态系统

Elasticsearch 生态系统（ecosystem）中有许多重要的组件，本节中将对其中一些重要的组件进行深入介绍。

10.10.1　Elasticsearch 分析器

Elasticsearch 以一种系统化的易于访问的方式来存储数据。为了让数据更易于分析和搜索，在数据被灌入 Elasticsearch 之前，需要进行以下步骤的处理：

1）最开始，对接收到的字符串进行清理。可以通过 Elasticsearch 中的字符过滤器（character filter）来实现。该过滤器能在字符串被分词之前进行清理。它也可以移除某些不必要的字符，甚至能对特定字符做变换处理。

2）分词（tokenize），将字符串转化为词项序列，词项用于构建倒排索引。分词操作可由 Elasticsearch 的分词器来进行处理。存在多种分词器来承担分词任务，即将字符串分隔为一个个词项或词条。

3）对数据或词项进行规范化（normalize），使搜索变得更简单，更相关。规范化可由 Elasticsearch 中的词条过滤器（token filter）来完成。过滤器既可以移除那些与搜索不相干的词条（如 a、the 等停用词），也可以对词条进行修改。

图 10-18 展示了分析器的工作原理。

Elasticsearch 提供了很多内置的字符过滤器、分词器、词条过滤器。这些组件可以按需任意组合使用，组合使用这些组件的对象被称为分析器（analyzer）。

存在两类分析器，如下所列：

- 内置分析器（built-in analyzer）
- 自定义分析器（custom analyzer）

仅供非商业用途或交流学习使用

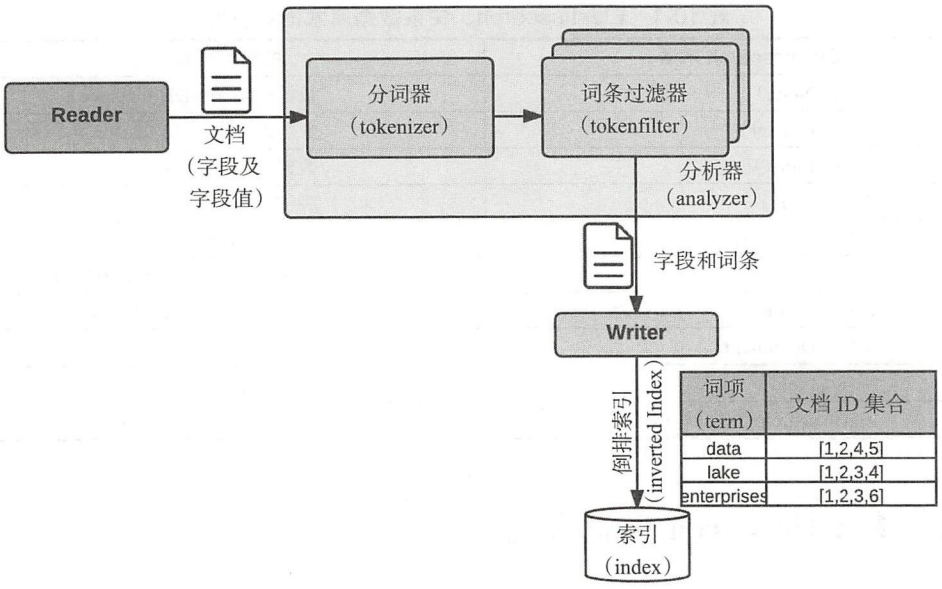


图 10-18 分析器的工作原理

1. 内置分析器

Elasticsearch 内置集成了很多分析器。对于一般的场景，这些分析器已经非常棒了。下面简要介绍一些最常见的分析器：

- Whitespace 分析器：基于空格对字符分词，生成的词条被添加到 Elasticsearch 中。
- Standard 分析器：如果没有显式指定分析器，它将作为默认分析器。对于大多数案例，Standard 分析器可用。该分析器可以用来处理任意语言的文本，将所有字符转换为小写来处理。
- Simple 分析器：任意非可读字符作为分隔符来产生词条，字符最终被转换为小写。
- Language 分析器：顾名思义，此类分析器用于处理特定语言的分析，它们能理解特定语言特性，并能对其进行处理。Elasticsearch 默认集成了很多 Language 分析器。

2. 自定义分析器

正如前面描述过的那样，如果内置分析器不够用，可以组合使用这些组件来构建自定义的分析器，以满足分析需求。

10.10.2 Elasticsearch 插件

Elastic Stack 在 X-Pack 中集成了多种有用的插件。此外，工业界还存在其他多种可用的插件，这些插件能增强 Elasticsearch 的处理能力。下面列举了一些外部插件：

- Head 插件：监控和统计 Elasticsearch 运行状态的经典插件，因 X-Pack 青睐 Kopf 插件而不再可用。

仅供非商业用途或交流学习使用

- Kopf 插件：监控和统计 Elasticsearch 运行状态的插件，作为 X-Pack 的一部分被集成在 Elastic Stack 中。
- BigDesk 插件：打包在 X-Pack 中，能生成 Elasticsearch 集群的实时图表和统计数据。
- Elasticsearch-SQL 插件：默认没有集成在 X-Pack 中，但是它提供 Elasticsearch 中的类 SQL 查询功能。
- Paramedic：默认打包在 X-Pack 中。

10.11　Elasticsearch 部署选项

Elasticsearch 支持水平扩展的部署架构，它优先考虑的是可用性。Elasticsearch 集群由 Master 节点和 Data 节点组成。基于 Quorum 机制的可用性策略一般来说能更好地保障集群的可用性。

虽然最常见的部署是单数据中心部署，它可以包含专用 Master 节点和专用 Data 节点，但对于较大的集群，也需要多个数据中心部署，以便提高重要应用服务的可用性。

跨数据中心部署 Elasticsearch 集群通常不是一个好主意，因为 Elasticsearch 领导者选举算法和 Data 节点的选择都是基于网络距离的，此时假定所有节点及其资源是等价的，因此在一个集群中，期望所有节点都是等距的。跨数据中心的集群中时常不能满足这个假设，因此不值得冒这个风险。

所以，通常建议在每个数据中心中都配置独立的集群，并且提供某种模式来保障数据可用性（同时从多个集群摄取数据）。该模式指的是任意时刻同时在两个集群中拉取数据，或者有一个跨数据中心镜像机制，具有类似 Kafka Mirror 这样的功能。

10.12　Elasticsearch Client

使用 Java 编程语言时，有以下方式可与 Elasticsearch 通信：

- Transport Client：Elasticsearch 在内部被 REST Client 使用。Elasticsearch 全力支持此类客户端，因为在系统内部利用它们来完成各种任务。这是最有效的通信方式，而且速度非常快。它使用二进制协议与 Elasticsearch 通信，在 Elasticsearch 内部也使用这种通信协议。下面的代码显示了如何获得 Transport Client 来与 Elasticsearch 通信。Transport Client 的更多细节可以在 Elastic 的官方博客 https://goo.gl/0ZKZIk 中找到。

```
TransportAddress address = new
InetSocketTransportAddress(InetAddress.getByName("localhost"),
9200);
Settings settings = Settings.builder().put("cluster.name",
"dataLakeCluster").build();
Client client = new
PreBuiltTransportClient(settings).addTransportAddress(address);
```

仅供非商业用途或交流学习使用

- Java REST Client：使用 HTTP 与 Elasticsearch 进行通信。它与大多数 Elasticsearch 版本兼容。其内部使用 Apache HTTP Async 客户端发送 HTTP 请求。Java REST Client 的更多细节可以在 Elastic 的官方博客 https://goo.gl/IKui5X 中找到。

```
RestClient restClient = RestClient.builder(new
HttpHost("localhost", 9200, "http")).build();
Response response = restClient.performRequest("GET", "/",
       Collections.singletonMap("pretty", "true"));
```

- Jest：除了默认的 REST Client，这是另一个基于 HTTP 的 Elasticsearch Client。下面的 Java 代码展示了如何获取 JestClient。

```
Client JestClientFactory factory = new JestClientFactory();
factory.setHttpClientConfig( new
HttpClientConfig.Builder("http://localhost:9200").multiThreaded(tru
e) .build());
JestClient client = factory.getObject();
```

- Spring Data Elasticsearch：Spring 是一个非常流行的 Java 框架。它通过一个名为 Spring Data 的模块屏蔽很多低层次数据存储细节。Spring 有一个特殊的模块，它将 Elasticsearch 从底层细节中抽象出来，即 Spring Data Elasticsearch 模块。

10.13 Elasticsearch 用于快速流式处理

数字时代，数据是核心，具备实时数据分析能力是所有企业的要求。在与其他流式处理框架（在我们的范例中为 Flink）协同处理实时数据时，Elasticsearch 起到了非常重要的作用。图 10-19 显示了此类数据处理的典型设置，并且与技术选择和用例实现非常相关。为实现图 10-19 中提及的架构，这里可以使用任意流式处理框架（Spark Streaming）来替代 Flink。

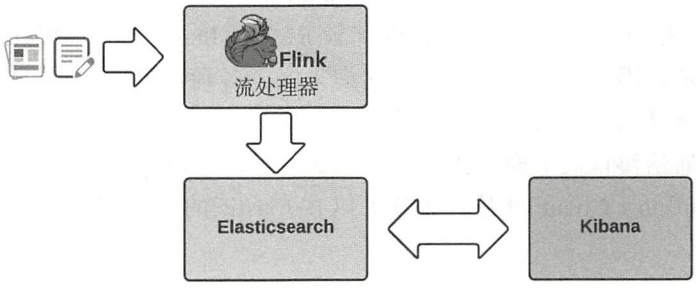


图 10-19 Elasticsearch 协同 Flink 处理流式数据

该架构是非常相关和有用的，因为使用 Flink 可以进行数据分析和转换，之后在服务层可使用 Elastic Stack 提供快速的数据查询服务。Kibana 是 Elastic Stack 的内置组件，能提供洞察 Elasticsearch 中数据的能力，它的功能非常完备。Kibana 能使可视化更加容易，而

非卖品，仅供非商业用途或交流学习使用

且完全不需要构建新的 UI。

在 Lambda 或 Kappa 架构中，Elasticsearch 技术组件在快速处理层和服务层中起到了非常重要的作用。服务层可以从 Hadoop（或 Cassandra 等其他数据存储系统）中获取批处理视图以及从 Elasticsearch 中获得实时视图，然后进行合并来构建企业应用程序所需的统一视图。

10.14　Elasticsearch 作为数据源

一般来说，Elasticsearch 不应被用作主要的数据存储系统（我们确实应该承认这一点）。然而，这个问题更多是用例驱动的，对于某些用例，Elasticsearch 可以很好地用作数据存储系统。Elasticsearch 属于 NoSQL 数据库，不支持典型关系型数据库的 ACID 特性，该特性主要用于面向事务（transaction-oriented）的用例。但它确实有一些特性，比如乐观锁（optimistic locking）和最终一致性（eventual consistency），这使得它适合于某些特殊的用例。对于一个数据湖实现，它可以很好地充当数据存储系统，因为真正的数据存储系统（记录系统）是与源系统在一起。Elasticsearch 出现任何失败，数据都可以很好地重新加载到 Elasticsearch 中（在实际的场景中，并没有这么直接），数据可以从源系统甚至从 Hadoop 加载，让 Elasticsearch 集群返回到工作状态（这显然依赖于数据湖的数据量）。

由于 Elasticsearch 的 NoSQL 特性，它确实提供了分布式、健壮性和无模式的能力，这使得它成为数据存储系统的一个不错的选择，但它并不是万能的，建议根据实际用例来评估它。

10.15　Elasticsearch 用于内容索引

许多内容管理系统（Content Management System，CMS）的典型问题是由于它们与一些数据存储紧密耦合而缺乏可扩展性以及性能低下，这里的数据存储主要是指传统 RDBMS。该问题可以通过将内容发布到更鲁棒的 NoSQL 存储系统来解决，Elasticsearch 是最强有力的竞争者之一。Elasticsearch 的核心特性是它的全文索引和搜索功能，这种能力是 CMS 所需的，同时，Elasticsearch 还具备强大的存储能力。Elasticsearch 也得到了市场上许多 CMSes(基于模板设计的网站内容管理系统）的支持，这使得它成为一个更合适的技术选择。出于这个原因和其他原因，CMS 和数字资产管理（Digital Asset Management，DAM）的许多平台都使用 Elasticsearch 作为内部的组件之一。

对以上话题的深入探讨超出了本章的范围，并会将我们带入一个与网站和内容管理相关的陌生领域。这里提及该话题的原因是想让读者理解 Elasticsearch 所具备的能力。

10.16　Elasticsearch 与 Hadoop

Hadoop 是大数据领域不能忽视的技术。Elasticsearch 对 Hadoop 有很好的支持，而

非卖品，仅供非商业用途或交流学习使用

Elastic Stack 中有一个名为 ES-Hadoop 的产品可实现这种集成。下面将详细讨论这个问题。想了解更多细节，建议读者参考 Elastic 官方网页 https://www.elastic.co/products/hadoop。

Hadoop 被认为适合批量分析，但是对于现代应用程序及企业来说，处理批量数据和实时数据时，执行有意义的分析是至关重要的。在处理实时数据时，Elasticsearch 和 Hadoop 的集成是恰当的。Elasticsearch 可以在 Hadoop 之上提供实时分析功能。ES-Hadoop 是一个由 Elastic 发布的集成在 Elastic Statck 中的 connector。

该 connector 与 Hadoop 1.x 及 2.x 兼容。因为它能使数据在 Hadoop 和 Elasticsearch 之间流动，因此它得到了很多大数据技术的青睐。

图 10-20 精确展示了 ES-Hadoop connector 的工作原理。

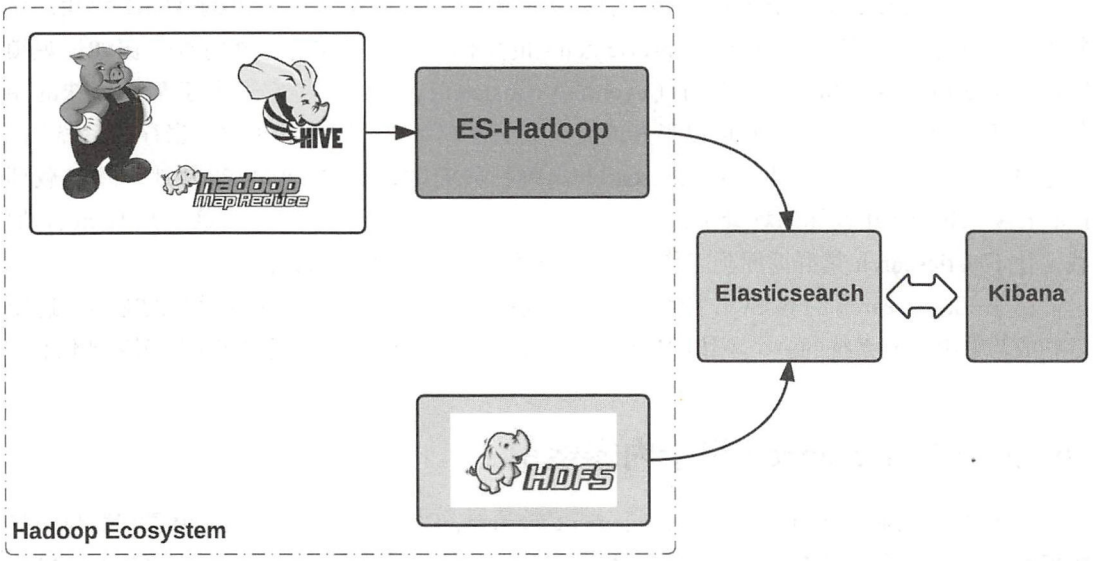


图 10-20 ES-Hadoop connector 工作原理（www.elastic.cc）

如图 10-20 所示，ES-Hadoop 已经为很多大数据技术提供了原生支持，允许从 Elasticsearch 读取 / 查询数据。ES-Hadoop 也拥有多种安全选项，可根据实际需求来确定使用何种安全选项。

对这个库的详细介绍超出了本节的范围，建议读者阅读下面的 Elastic 文档：https://goo.gl/NSwCVE，以获得关于该库更深入、全面的信息，ES-Hadoop 是实现数据湖所必需的。

10.17 Elasticsearch 运行范例

在本章中，已经了解了 Elasticsearch 的各种功能，现在我们将尝试通过运行范例来查看它的一些主要功能。首先从安装 Elastic Stack 开始。

非卖品，仅供非商业用途或交流学习使用

10.17.1　安装

1）下载 Elastic Stack 5.x 安装包，可使用下面的命令从指定位置获取：

```
wget
https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-
5.3.0.tar.gz
```

2）切换至某用户目录，使用下面的命令对 Elastic Stack 5.x 安装包解压：

```
tar -zxvf ${DOWNLOAD_DIR}/elasticsearch-5.3.0.tar.gz
```

3）可使用 ${ES_HOME} 来引用解压目录，将其设置为环境变量，如下所示，最后将同样的内容追加到 ~/.bashrc 文件中：

```
export ES_HOME=<path-to-elasticsearch-folder>
export PATH=$PATH:$ES_HOME/bin
```

4）编辑 ${ES_HOME}/config/elasticsearch.yml 文件，如下所示：

```
network.host : 0.0.0.0
```

5）确保下列限制在 /etc/security/limits.conf 中已配置。如果这些限制都没有配置，则组合使用 sudo 及 vi 命令对该文件进行编辑，配置范例如下：

```
sudo vi /etc/security/limits.conf

*    softnproc65536
*    hard nproc 65536
*    soft nofile 65536
*    hard nofile 65536
```

6）通过 root 账户（su -root）编辑 /etc/sysctl.conf 文件来改变系统变量 vm.max_map_count 的值，将下面这行内容追加到文件尾部：

```
vm.max_map_count = 262144
```

前面的配置是为了增加 Linux 环境中运行的进程的资源可用性。从 Elasticsearch 5.x 开始，为了成功启动 Elasticsearch 进程，这些修改是强制的。

7）重启服务器使之前的配置生效，在重启服务器之前，请确保重要的进程已妥善关闭，然后运行下面的命令来启动 Elastic 进程：

```
${ES_HOME}/bin/elasticsearch
```

前面的步骤中，按默认配置启动了 Elasticsearch（即启动 Master 节点及 Data 节点）。这些步骤对各个 Elasticsearch 版本来说，是通用且一致的。

为了完成本章中列出的大多数范例，将使用 Sense 插件，该插件运行于 Kibana 之上。因此，下一步将运行安装了 Sense 插件的 Kibana。

Kibana 的安装包是操作系统（OS）相关的。对于 Linux 平台，可以为 tarball 格式，也

非卖品，仅供非商业用途或交流学习使用

可以是 RPM 格式，同时提供了 32 位和 64 位版本。出于演示本书范例的目的，我们选用了 tarball 格式的 64 位 Linux 安装包。请参考下面的步骤：

1）使用下面的命令从目标位置下载 tarball 文件：

```
wget
https://artifacts.elastic.co/downloads/kibana/kibana-5.3.0-linux-x8
6_64.tar.gz
```

2）切换到某个用户目录，并使用下列命令提取 tarball 中的内容：

```
tar -xzvf ${DOWNLOAD_DIR}/kibana-5.3.0-linux-x86_64.tar.gz
```

3）使用下面的命令设置环境变量 ${ KIBANA_HOME }，使之指向 Kibana 解压目录。并将它们追加到 ~/.bashrc 文件中去。

```
export KIBANA_HOME=<path-to-kibana-directory>
export PATH=$PATH:$KIBANA_HOME/bin
```

4）编辑 ${KIBANA_HOME}/config/kibana.yml 文件，使用下面的设置来配置远程连接：

```
#为了允许远程用户连接，将这个参数设置为anon-loopback地址
 server.host: 0.0.0.0
```

5）用下面的命令启动 Kibana：

```
${KIBANA_HOME}/bin/kibana
```

上面的命令启动了 Kibana，在浏览器中输入 http://localhost:5601 将能看到登录页（localhost 也可以用 IP/ 机器名替换，实现远程访问），下面将展示登录页。Dev Tools（左侧菜单）中提供了一个与 Elasticsearch 服务器交互的控制台。这就是大家所熟知的 Sense 插件，直到 Elasticsearch 2.x 之前，都需要为这些版本独立安装 Sense 插件。

我们简单地提到了 Elasticsearch 2.x，这是因为有些框架（如 Flink）还没有完全支持 Elasticsearch 5.x。因此，当与 Flink 整合时，可能需要切回到 Elasticsearch 2.x 版本。不过值得欣慰的是，书中使用到的范例对 Elasticsearch 5.x 或 Elasticsearch 2.x 来说是一样的，唯一不同的是它的安装和设置。

如图 10-21 所示为 Kibana 的登录页面。

Kibana 5.x 提供了 Dev Tools 控制台，可以在这里演示所有的范例，如图 10-22 所示。GET _search 命令旁边的小按钮用于执行查询，结果显示在右侧窗口中。

现在已经安装了 Elasticsearch 和 Kibana。这就完成了运行范例所需的工具集了。接下来将介绍 Elasticsearch 实战的相关内容。

非卖品，仅供非商业用途或交流学习使用

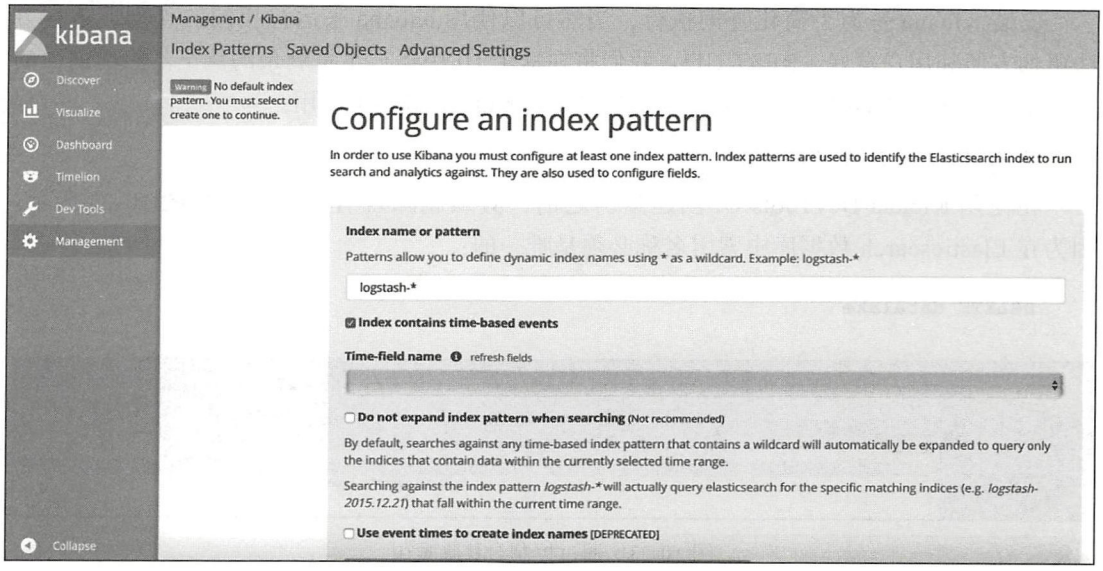


图 10-21　Kibana 登录页面

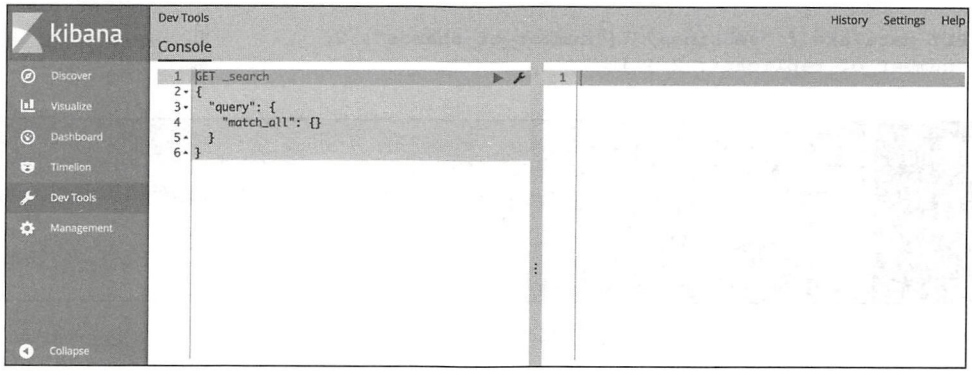


图 10-22　Kibana 用于执行查询的控制台

10.17.2　创建和删除索引

可通过以下命令（curl）来创建 Elasticsearch 集群中的索引，该命令可在 Linux ssh 控制台中运行：

1）请求：

```
curl -XPUT http://<es-server-ip>:9200/datalake -d
'{"settings":{"number_of_shards": 2, "number_of_replicas": 2}}'
```

2）响应：

```
{"acknowledged":true,"shards_acknowledged":true}
```

非卖品，仅供非商业用途或交流学习使用

安装 Kibana 是为了简化查询执行，当然可以使用 Kibana 来运行相同的查询。这种方法更加方便和用户友好（REST UI）。我们将在这个 UI 中运行本章剩余的例子。然而前面的内容表明，简单的 curl 命令也可以在没有安装 Kibana 的机器中使用。

使用 Kibana Dev Tools UI

在使用 Kibana DevTools UI 创建索引之前，必须删除现有的索引，如图 10-23 所示。因为在 Elasticsearch 数据库中索引名称必须是唯一的：

```
DELETE datalake
```

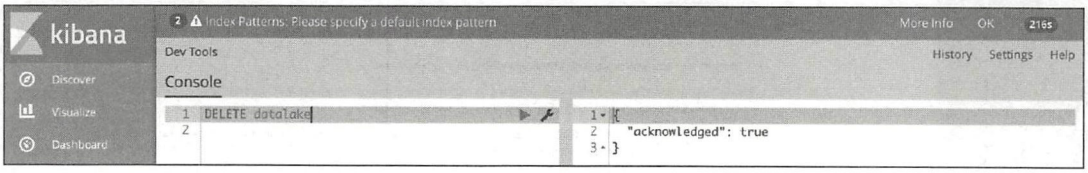


图 10-23 通过控制台移除索引

现在，使用以下查询再次创建相同的索引（参见图 10-24）：

```
PUT datalake { "settings" :{"number_of_shards": 2,
"number_of_replicas": 2 } }
```

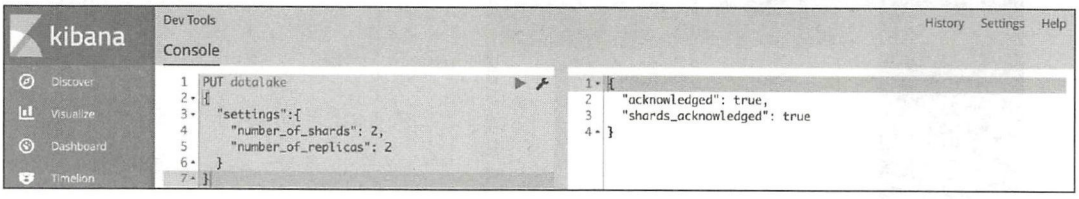


图 10-24 索引创建 Query

10.17.3 对文档进行索引

现在已经创建了索引，可以将文档添加到索引中。对文档进行索引的唯一要求是，它是一个 JSON 文档，因为 Elasticsearch 是无模式（schema-less）的，并且基于被索引文档的结构推导出存储模式。

如图 10-25 所示，索引文档的命令为 PUT{indexname}/{type}/{id}：

```
PUT datalake/contacts/101
{
  "id":101,
  "cell":"(478) 531-2026",
  "work":"1-906-774-1226",
  "email":"vincenzo.hickle@yahoo.com"
}
```

非卖品，仅供非商业用途或交流学习使用

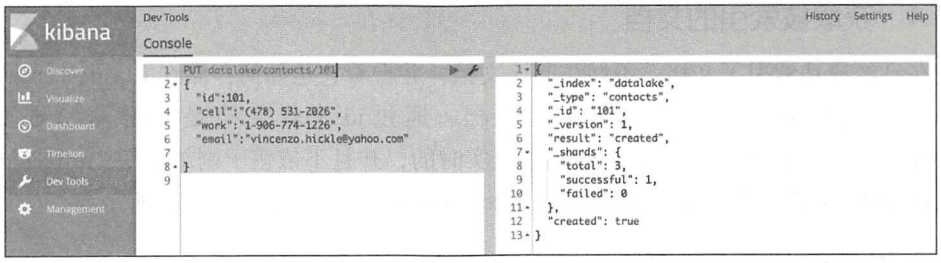


图 10-25　文档创建 Query

随着文档的索引，Elasticsearch 在内部基于初始文档中数据创建文档映射（mapping）。这个映射可以被访问，如图 10-26 及以下命令所示：

```
GET datalake/_mapping/contacts
```

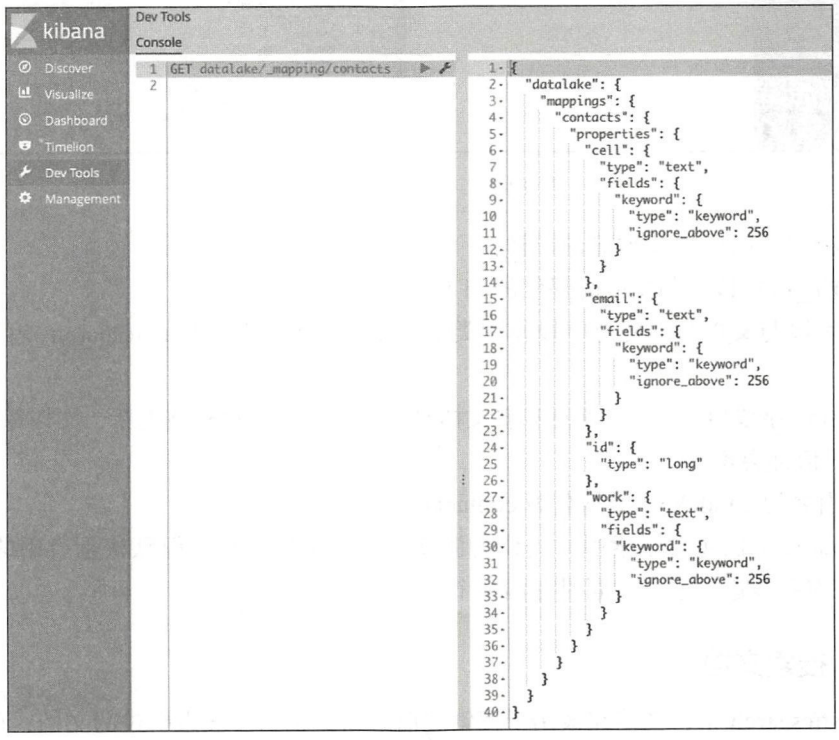


图 10-26　通过 Query 查询文档映射

可从图 10-26 中观察到一些现象：

- Keyword 分析器为默认的无模式索引分析器。
- 根据输入类型（type），模式中的类型信息可以被自动推导出来。
- 针对文本字段，已经设置了一些默认的验证操作。

这些特征非常强大，它们能提供对结构化或半结构化数据的灵活的索引能力。

非卖品，仅供非商业用途或交流学习使用

10.17.4 获取被索引的文档

现在已经成功索引了文档，不妨访问该文档，并看看它是如何存储在 Elasticsearch 中的。

为了完成该任务，可通过 id 访问文档。通过 id 获取文档是一种非常重要的从 Elasticsearch 中访问文档的方法，因为它总是实时的，并且不依赖于刷新周期（refresh cycle）。

通过 id 访问文档的一般语法是 Get{ index-name }/{type}/{id}，如下所示：

```
GET datalake/contacts/101
```

命令运行效果如图 10-27 所示。

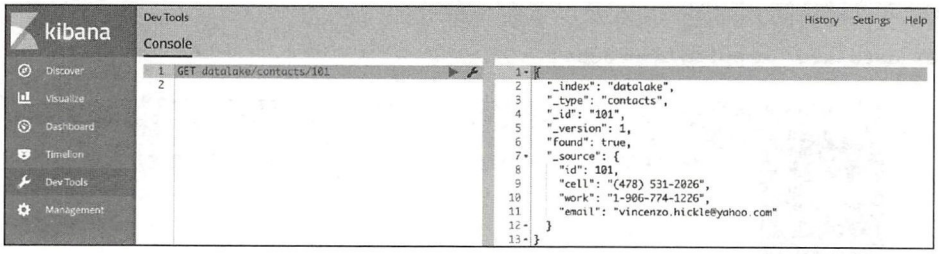


图 10-27 通过 id 查询文档

正如我们在图 10-27 中看到的那样：

1）文档 _id 字段值与索引时提供的 id 相同。

2）id 字段与文档其他任意字段作用类似，不像 _id 字段是 Elasticsearch 专门用于查找文档的。

3）version 字段值等于 1，这意味着文档刚刚被创建，还没有被更新，文档每次被更新，version 字段值都会累加。

4）索引名为 datalake，类型名为 contacts。

5）_source 字段包含了索引时提供的源文档。该字段可选，但默认是启用的。该字段的作用是在对字段进行索引的同时存储整个源文档。

10.17.5 搜索文档

在 Elasticsearch 中，文档在被用于搜索之前必须先索引。被索引过的文档，可以通过 _search url 被下列任意 Query（每种 Query 有其特殊的用途）搜索，例如：

1. GET {index-name}/_search

下面是搜索索引中所有文档的范例：

```
GET datalake/_search
```

命令运行结果如图 10-28 所示。

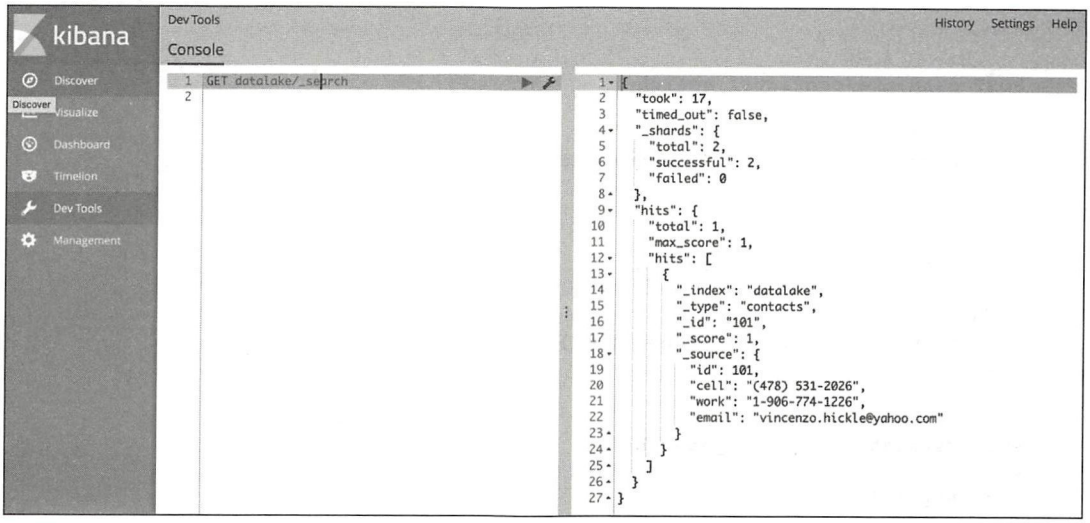


图 10-28　通过 _searchURL 获取索引中所有文档的 Query

2. GET {index-name}/{type}/_search

现在给出的是搜索索引中某个类型下所有文档的例子：

```
GET datalake/contacts/_search
```

命令运行结果如图 10-29 所示。

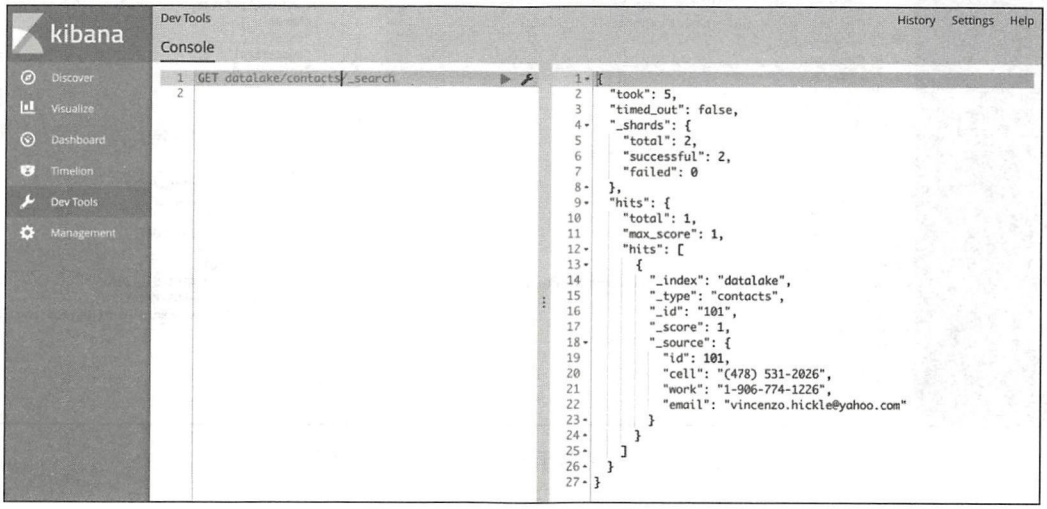


图 10-29　获取索引中某个类型下所有文档的 Query

3. POST {index-name}/{type}/_search

如果想使用搜索参数来搜索某个索引特定类型中的所有文档，需要使用 POST 命令，

仅供非商业用途或交流学习使用

如下所示：

```
POST datalake/contacts/_search
{
  "query": {
    "term": { "email" : "vincenzo.hickle"}
  }
}

POST datalake/contacts/_search
{
  "query": {
    "term": { "email" : "yahoo.com"}
  }
}

POST datalake/contacts/_search
{
  "query": {
    "term": { "email" : "vincenzo.hickle@yahoo.com"}
  }
}
```

图 10-30 中给出了几种基于 term 的搜索模式变种。

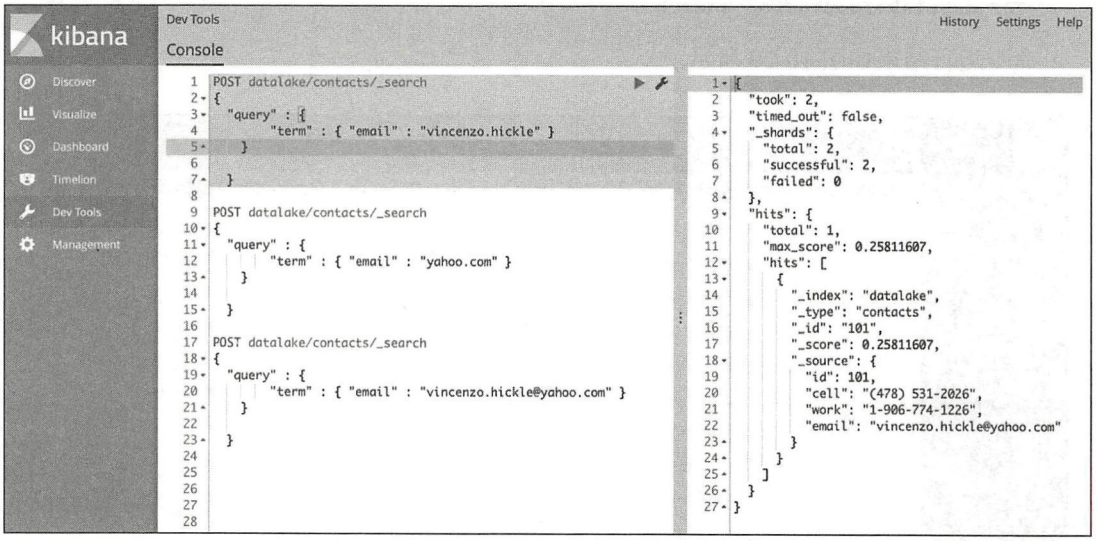


图 10-30 几种基于 term 的搜索模式变种

如读者所见，有 3 种不同的 POST Query，可以观察到下列现象：

- 第 1 个查询视图找到匹配的 E-Mail 地址，然后得到一个匹配部分 E-Mail 地址的命中结果。
- 第 2 个查询也试图找到匹配的 E-Mail 地址，只是匹配的是 E-Mail 地址的另一部分。

仅供非商业用途或交流学习使用

- 第 3 个查询视图查找完全匹配的 E-Mail 地址，此时返回记录条数为零，因为索引中不存在对应的词条。

第 3 个查询没有命中文档，这是由作用在 email 字段上的 keyword 分析器的特性决定的。如果想让 email 字段完全匹配 E-Mail 地址，则需要对 email 字段按 multi-index 方式进行处理（对一个字段索引时使用多种分析器）。

除了上面提到的多种匹配模式，_search URL 还支持多种额外的功能，如分页（pagination）、排序、聚合（aggregation）、多重匹配（multiple match）功能等。

10.17.6　更新文档

现在来尝试更新同一个文档，并修改电子邮件地址。文档更新命令形式为 POST {index-name}/{type}/{id}，请求体范例如下所示：

```
POST datalake/contacts/101
{
  "id":101,
  "cell":"(478) 531-2026",
  "work":"1-906-774-1226",
  "email":"vincenzo2.hickle@yahoo.com"
}
```

命令运行结果如图 10-31 所示。

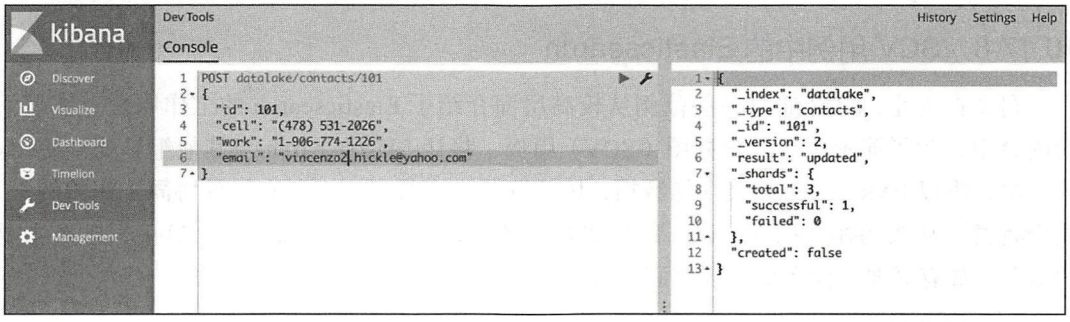


图 10-31　文档更新 Query

从前面的查询响应中可以观察到以下现象：

- version 字段值被更新为 2。
- created 标记被设置为 false，result 字段值暗示已经成功更新文档。
- 总共有 3 个分片受影响，因为最初索引被配置为拥有 2 个副本和 4 个分片。这意味着有 3 个分片中的数据受修改的影响，其中一个是主分片，其他两个为副本。

10.17.7　删除文档

可使用 DELETE 命令执行文档删除操作，其格式为 DELETE {indexname}/{type}/{id}，

仅供非商业用途或交流学习使用

范例如下所示：

```
DELETE datalake/contacts/101
```

命令运行结果如图 10-32 所示。

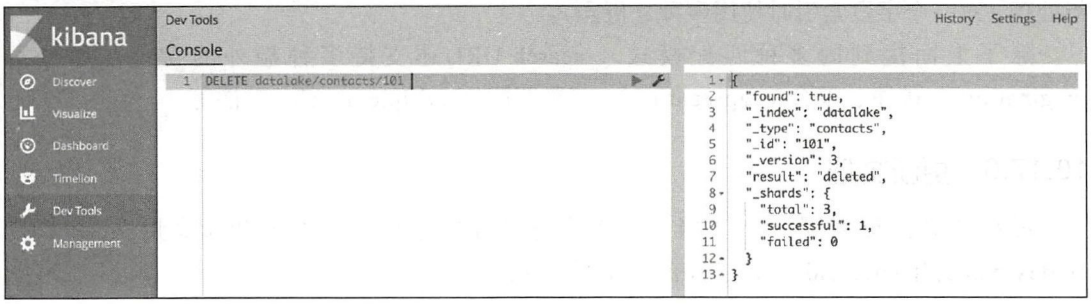


图 10-32 文档删除 Query

如结果所示，有一个文档被删除。在 Elasticsearch 中通常是按 id 来执行删除的；也支持按查询来删除，但这是一个更费时的操作。与前面的例子类似，也有 3 个分片被影响，理由是相同的。

值得注意的是，这里的删除并不是立即进行物理删除，直到底层进行索引优化和压缩时，逻辑上被删除的文档才真正被物理删除。

10.17.8 SCV 用例中的 Elasticsearch

到现在为止，已经通过一些范例从较高层面介绍了 Elasticsearch 的工作原理。在下面的内容中，为了演示单一客户视图（SCV）用例，将让 Elasticsearch 与其他组件配套使用。这一次，将以 PARQUET 格式存储数据，因为这是本书最后一个所有组件协同工作的例子，我们将尝试从头构建数据湖，以了解从数据准备开始的一系列集成操作。这里只涉及主要的命令，如有必要，读者请阅读前面章节以了解更多细节。

1. 数据准备

范例数据使用了与之前相同的数据集，有 200 万条客户记录，包括住址、客户联系方式。

首先通过下面介绍的步骤来清理前几章中创建的数据。确保所需的进程已经启动并运行，然后进行清理工作。所需的进程包括 Hue、DFS、hiveserver2、ZooKeeper 和 Kafka。

（1）初始化（清理）

使用下面的命令来删除 Hive 元数据中的表，如果有任何其他多余的表，也请一并删除：

```
drop table customer;
drop table address;
drop table contacts;
```

仅供非商业用途或交流学习使用

停止 dfs 服务（运行 stop-dfs.sh），通过格式化 NameNode 来清除 Hadoop 存储，参考下面的命令：

```
${HADOOP_HOME}/bin/hdfs namenode -format
```

使用下面的命令创建新的 Hadoop 目录：

```
hdfs dfs -mkdir -p /datalake/raw/customer
hdfs dfs -mkdir -p /datalake/raw/address
hdfs dfs -mkdir -p /datalake/raw/contact
```

移除 Kafka 中所有 topic 以确保启动时没有历史数据，参考下面的命令：

1）获取 topic 列表：

```
${KAFKA_HOME}/bin/kafka-topics.sh --list --zookeeper 0.0.0.0:2181
```

2）对于每个 topic，执行下面的删除命令（下划线前缀的 topic 除外）：

```
${KAFKA_HOME}/bin/kafka-topics.sh --delete --topic <topic-name> --
zookeeper 0.0.0.0:2181
```

（2）数据生成

使用下面的命令生成数据，这与前面章节中的做法相同。在当前的范例中，我们需要为每个实体生成 100 万条记录：

Populate database

这个例子中会重新使用 customer 和 address 数据库表。如果由于某种原因数据库表为空，可使用与以前相同的应用程序重新填充数据库。这里再次展示了相关的命令，以便参考。该命令需要从解压目录中运行：

```
java -jar client-generator-1.0-SNAPSHOT.jar config/db.properties  2000000
```

（3）生成 spool 文件

也可以重用之前生成的 contacts.log 文件。如果该文件不存在，则重新生成它，具体细节参考前面章节中的例子。读者可参考下面生成数据的命令，它必须在生成器（generator）所在目录中运行：

```
java -jar web-generator-1.0-SNAPSHOT.jar contact.log 200000 contact
```

一旦 spool 文件可用，将 spool 文件移动到 ${FLUME_DATA} 目录中供 Flume 处理。

2. 使用 Sqoop 将 customer 数据导入 Hive

下面的配置可能需要被添加到 ${HADOOP_HOME}/etc/hadoop/core-site.xml 文件中，这样 Hue 可以模拟用户创建 Parquet 文件：

仅供非商业用途或交流学习使用

```
<property>
  <name>hadoop.proxyuser.hue.hosts</name>
  <value>*</value>
</property>
<property>
  <name>hadoop.proxyuser.hue.groups</name>
  <value>*</value>
</property>
```

添加前面的配置后，可重启 dfs 服务，参考下面的命令：

```
${HADOOP_HOME}/sbin/stop-dfs.sh && ${HADOOP_HOME}/sbin/start-dfs.sh
```

使用 Sqoop job，将从数据库加载 customer 记录至 Hadoop 中，写入的文件为 Parquet 格式，参考下面的命令：

```
${SQOOP_HOME}/bin/sqoop import --connect
jdbc:postgresql://<DB_SERVER_ADDRESS>/sourcedb?schema=public --table
customer --m 1 --username postgres --password <DB_PASSWORD> --as-
parquetfile --append --target-dir /datalake/raw/customer/$(date +%Y-%m-%d--
%H-%M)
```

从前面的命令中可以观察到以下现象：

- 使用 Parquet 文件存储 customer 信息。
- 只使用了一个 mapper (--m 1)。
- 以分钟为单位对 customer 数据进行切割（$(date +%Y-%m-%d--%H-%M)）。

输出的 Parquet 文件可以通过 snappy 编解码器（codec）来进行压缩，不过需要提供额外的参数：--compression-codec。出于一致性方面的考虑，本范例进一步会以分钟为单位，对 HDFS 上的数据进行分割，如图 10-33 所示。

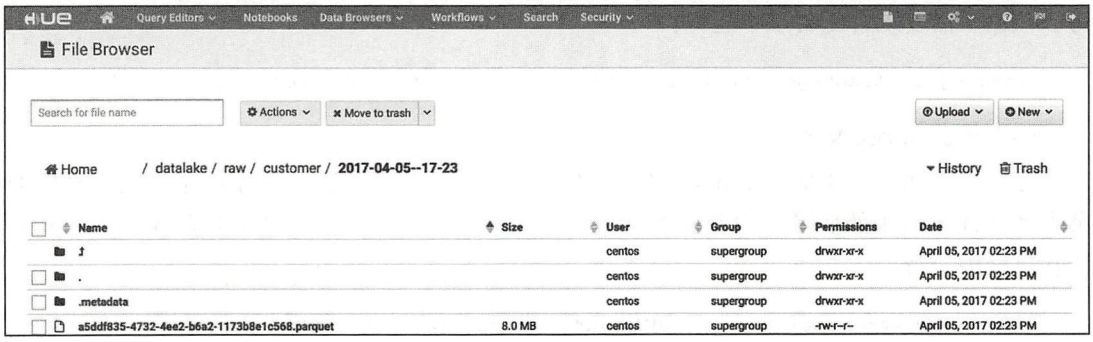


图 10-33 利用 Sqoop 将 Customer 数据导入 HDFS（Parquet 文件）

3. 利用 Flume 进入 Kafak Channel 获取数据

现在使用 Flume 从数据库及 spool 文件中获取 address、contact 数据（处理方式与前一章相同）。为了实现该目标，需定义一个 Flume 配置文件 ${FLUME_HOME}/conf/customer-

仅供非商业用途或交流学习使用

address-contact-conf.properties，需要对 Kafka Channel 进行适当配置，使之能将两个数据源中的数据转换为事件（events）。

下面是完整的 Flume 配置文件：

```
agent.sources = sql-source spool-source

agent.sources.spool-source.type=spooldir
agent.sources.spool-source.spoolDir=<spool-file-data-dir>
agent.sources.spool-source.inputCharset=ASCII

agent.sources.sql-source.type=org.keedio.flume.source.SQLSource
agent.sources.sql-
source.hibernate.connection.url=jdbc:postgresql://localhost/sourcedb?schema
=public
agent.sources.sql-source.hibernate.connection.user=postgres
agent.sources.sql-source.hibernate.connection.password=<db-password>
agent.sources.sql-
source.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
agent.sources.sql-
source.hibernate.connection.driver_class=org.postgresql.Driver
agent.sources.sql-source.custom.query= SELECT id, customer_id, street1,
street2, city, state, country, zip_pin_postal_code from address WHERE id >
$@$ ORDER BY id ASC
agent.sources.sql-source.status.file.path=/data1/var/lib/flume/customer-db
agent.sources.sql-source.status.file.name=<path-of-status-file>
agent.sources.sql-source.batch.size = 5000
agent.sources.sql-source.max.rows = 5000
agent.sources.sql-source.read.only=true
agent.sources.sql-source.hibernate.c3p0.min_size=10
agent.sources.sql-source.hibernate.c3p0.max_size=10
agent.sources.sql-source.hibernate.connection.provider_class =
org.hibernate.connection.C3P0ConnectionProvider

# The channel can be defined as follows.
agent.channels = contactChannel addressChannel

agent.channels.contactChannel.type
=org.apache.flume.channel.kafka.KafkaChannel
agent.channels.contactChannel.kafka.bootstrap.servers=<kafka-broker-ip-
addr>:9092
agent.channels.contactChannel.kafka.topic=contacts
agent.sources.spool-source.channels = contactChannel

agent.channels.addressChannel.type
=org.apache.flume.channel.kafka.KafkaChannel
agent.channels.addressChannel.kafka.bootstrap.servers=<kafka-broker-ip-
addr>:9092
agent.channels.addressChannel.kafka.topic=address
agent.sources.sql-source.channels = addressChannel
```

如上面配置文件所示，address 数据被转化为事件流，流入 address topic 中，contact spool 文件中的数据也被转化为事件流，流入 contact topic 中。前面的配置与我们迄今为止看到的 sql-source 的配置略有不同，因为 sql-source 中的数据在数据量较大时通常会有大量

仅供非商业用途或交流学习使用

重复记录。在这里，为 sql-source 添加了自定义查询（查询包含了特殊字符串“$@$”），它充当跟踪器，有助于防止记录的重复。

以前面的配置文件路径为参数，启动 Flume Agent，参考下面的命令：

```
${FLUME_HOME}/bin/flume-ng agent --conf ${FLUME_HOME}/conf/  -f
${FLUME_HOME}/conf/customer-address-contact-conf.properties -n agent -
Dflume.root.logger=INFO,console
```

分别来自数据库及 Spool 文件的 address、contact 数据被转化为事件流，分别被推送到对应的 Kafka topic 中去。

处理进度可以通过查看 topic 的 offset 来了解，参考以下命令：

```
${KAFKA_HOME}/bin/kafka-run-class.sh kafka.tools.GetOffsetShell --broker-
list <broker-ip-address>:9092 --topic address

${KAFKA_HOME}/bin/kafka-run-class.sh kafka.tools.GetOffsetShell --broker-
list <broker-ip-address>:9092 --topic contacts
```

4. 使用 Flink 摄取数据至 HDFS、Elasticsearch 中

当事件流流入对应的 Kafka topic，Flink 处理流水线被触发，然后开始消费 Kafka topic 中的事件。

可参考前面章节中 Flink 的例子，我们在这里构建两条流水线，一条用于处理 address 数据，另一条用于处理 contact 数据。它们分别让事件流入对应的 Sink，对应的是 HDFS 与 Elasticsearch，因此这两类情况都是同一事务的一部分。

在前面的 Flink 示例中，我们是从 IDE 中运行，现在将尝试对范例打包，这样可以将代码部署到 Flink 容器中。这种 Flink 部署方式较为新颖，在前面的章节中没有提到。

为了实现这一点，不妨看看 Flink 流水线工程相关的关键元素。该工程的完整的源代码可以在 chapter10/speed-address-flink-ingestor（address Flink pipeline）和 chapter10/speed-contacts-flink-ingestor（contacts Flink pipeline）中找到。我们将逐节讨论所有代码来帮助读者理解 Flink 是如何摄取数据的。

（1）利用 POM 文件打包

下面显示了 POM 文件的结构，其中包括所有必需的依赖性。此外，它还配置了 Shade 和 Avro 插件，这有助于将所有依赖项打包成一个可自己执行的 JAR。Avro 插件基于 Avro 模式（schema）生成 Avro 类，该任务作为 maven 构建的一部分，以下是 POM 文件内容：

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<parent>
…..PARENT POM LINKAGE…..
</parent>
```

仅供非商业用途或交流学习使用

```
<groupId>com.laketravels</groupId>
<artifactId>speed-address-flink-ingestor</artifactId>
<dependencies>
<dependency>
<groupId>org.apache.kafka</groupId>
<artifactId>kafka-clients</artifactId>
<version>0.10.1.1</version>
</dependency>
 …..OTHER DEPENDENCIES…..
</dependencies>
<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-shade-plugin</artifactId>
<version>3.0.0</version>
<executions>
<execution>
<phase>package</phase>
<goals>
<goal>shade</goal>
</goals>
<configuration>
<artifactSet>
<excludes>
<exclude>org.apache.flink:flink-shaded-*</exclude>
<exclude>org.apache.flink:flink-core</exclude>
 …..OTHER EXCLUDES THAT ARE PROVIDED BY FLINK CONTAINER…..
</excludes>
</artifactSet>
<filters>
<filter>
<artifact>org.apache.flink:*</artifact>
<excludes>
 …..FOLDER EXCLUDES FOR FLINK SHADED AND WEB-DOCS…..
</excludes>
</filter>
<filter>
<!- EXCLUDE SIGNATURES -->
<artifact>*:*</artifact>
<excludes>
<exclude>META-INF/*.SF</exclude>
 …..OTHER SIGNATURE EXCLUDES…..
</excludes>
</filter>
</filters>
<transformers>
 …..TRANSFORMERS TO INCLUDE MAIN CLASS AND RESOURCE TRANSFORMER…..
<createDependencyReducedPom>false</createDependencyReducedPom>
</configuration>
</execution>
</executions>
</plugin>
<plugin>
<groupId>org.apache.avro</groupId>
<artifactId>avro-maven-plugin</artifactId>
<version>1.7.7</version>
```

仅供非商业用途或交流学习使用

```
<executions>
<execution>
 …..AVRO PLUGIN CONFIGURATION FOR GENERATE SOURCE PHASE WITH TARGET CLASS
GENERATION FOLDERS…..
</execution>
</executions>
</plugin>
</plugins>
</build>
</project>
```

（2）Avro 模式定义

Parquet 文件格式使用了 Avro 模式来定义对象。这些 Avro 模式都存在于工程的 src/main/resources 目录中。就描述 address 而言，下面是典型的 Avro 模式：

```
{
   "namespace": "example.avro",
   "type": "record",
   "name": "Contact",
   "fields": [
      {"name": "id", "type": "string"},
      {"name": "customerId",  "type": "string"},
      {"name": "street1", "type": "string"},
      {"name": "street2", "type": "string"},
      {"name": "city", "type": "string"},
      {"name": "state", "type": "string"},
      {"name": "country", "type": "string"},
      {"name": "zipCode", "type": "string"}
   ]
}
```

（3）模式反序列化类

正如前面章节中介绍过的那样，这个类用于将 Kafka 事件转换为 Tuple2，这样，事件中的数据可以写入 HDFS 中。一旦事件被转换为 Tuple2，也可以被写入 Elasticsearch（ES）中。由于我们希望在同一个事务中处理 HDFS 及 ES Sink，这样的数据转换是必要的。

事实上，address、contact 的反序列化有细微的区别，因为两者源自不同的数据流。读者可以从 Flume 的配置中看到，address 数据来自数据库（sql-source），它在数据库记录之上添加了一个时间戳，字段值是由逗号分隔的。而 contact 数据源自 spool 文件，它在每条 Flume 事件之前插入了两个字节的信息，每条 contact 记录中包含了 spool 文件中的一行，字段值由逗号分隔。由于 address、contact 数据本身存在差别，因此两者的反序列化也存在相应的细微差别。

以下是 Address 对象的反序列化模式类代码：

```
public class Tuple2CustomerAddressDeserializationSchema implements
DeserializationSchema {

  private static final ObjectMapper MAPPER = new ObjectMapper();
  private static final int KAFKA_TIMESTAMP_LENGTH = 28;

  public Object deserialize(byte[] bytes) throws IOException {
```

```
        String message = new String(bytes,KAFKA_TIMESTAMP_LENGTH,
                                    bytes.length-(KAFKA_TIMESTAMP_LENGTH));
        if (message.trim().length()>0) {
            Address address = new Address();
            String[] fields = message.replaceAll("""","").split(",");
            address.setId(Integer.parseInt(fields[0].trim()));
            address.setCustomerId(Integer.parseInt(fields[1].trim()));
            address.setStreet1(fields[2]);
            address.setStreet2(fields[3]);
            address.setCity(fields[4]);
            address.setState(fields[5]);
            address.setCountry(fields[6]);
            address.setZipCode(fields[7]);
            Tuple2<IntWritable, Text> tuple = new Tuple2<IntWritable,
Text>();
            tuple.setFields(new IntWritable(address.getId()),
                    new Text(MAPPER.writeValueAsString(address)));
            return tuple;
        } else {
            return null;
        }
    }

    public boolean isEndOfStream(Object o) {
        return false;
    }

    public TypeInformation<Tuple2<IntWritable, Text>> getProducedType() {
        return new TupleTypeInfo<Tuple2<IntWritable, Text>>
           (TypeExtractor.createTypeInfo(IntWritable.class),
           TypeExtractor.createTypeInfo(Text.class));
    }
}
```

以下是 Contact 对象的反序列化模式类的代码：

```
public class Tuple2CustomerContactDeserializationSchema implements
DeserializationSchema {

    private static final ObjectMapper MAPPER = new ObjectMapper();

    public Object deserialize(byte[] bytes) throws IOException {
        //待处理消息的前两个字节为 Unicode 字符
        String message = new String(bytes, 2, bytes.length-2);
        if (message.trim().length()>0) {
            String[] contactFields = message.split(",");
            Contact contact = new Contact();
            contact.setId(Integer.parseInt(contactFields[0].trim()));
            contact.setCell(contactFields[1]);
            contact.setWork(contactFields[2]);
            contact.setEmail(contactFields[3]);
            Tuple2<IntWritable, Text> tuple = new Tuple2<IntWritable,
Text>();
            tuple.setFields(new IntWritable(contact.getId()),
                    new Text(MAPPER.writeValueAsString(contact)));
            return tuple;
        } else {
```

```
            return null;
        }
    }

    public boolean isEndOfStream(Object o) {
        return false;
    }

    public TypeInformation<Tuple2<IntWritable, Text>> getProducedType() {
        return new TupleTypeInfo<Tuple2<IntWritable, Text>>
            (TypeExtractor.createTypeInfo(IntWritable.class),
            TypeExtractor.createTypeInfo(Text.class));
    }
}
```

（4）向 HDFS 中写入 Parquet 文件

由于想将 Parquet 文件写入 HDFS，所以我们将使用之前用过的 BucketSink，同时使用了一个自定义 Parquet Writer 以及一个基于分钟时间单位分割的 DateTimeBucketer，如下所示。Bucket Sink 路径通过命令行参数 hdfsPath 来传递，稍后将讨论它。更新 HADOOP_USER_NAME 系统属性，在代码中将其赋值为 CentOS 用户账号：

```
System.setProperty("HADOOP_USER_NAME", "centos");

//HDFS Sink
BucketingSink<Tuple2<IntWritable, Text>> hdfsSink = new
BucketingSink<Tuple2<IntWritable,
Text>>(parameterTool.getRequired("hdfsPath"));
hdfsSink.setBucketer(new DateTimeBucketer("yyyy-MM-dd--HH-mm"));
hdfsSink.setWriter(new SinkParquetWriter<Tuple2<IntWritable,
Text>>("address.avsc"));
hdfsSink.setBatchSize(1024 * 1024 * 1);
messageStream.addSink(hdfsSink);
```

自定义的 Parquet Writer 将一个特定的数据对象作为 Avro 记录写入 HDFS，然后使用 AvroParquet 类将其转换为 Parquet 格式，并启用了 SNAPPY 压缩，这里操作的是 address 数据对象，如下所示：

```
private static class SinkParquetWriter<T> implements Writer<T> {

transient ParquetWriter writer = null;
    String schema = null;
transient Schema schemaInstance = null;
final ObjectMapper MAPPER = new ObjectMapper();

public SinkParquetWriter(String schema) {
this.writer = writer;
this.schema = schema;
try {
this.schemaInstance = new Schema.Parser().parse(getClass().getClassLoader()
                    .getResourceAsStream(schema));
        } catch (IOException e) {
throw new RuntimeException(e);
        }
```

```
    }

public void open(FileSystem fileSystem, Path path) throws IOException {
writer = AvroParquetWriter.builder(path)
                .withSchema(this.schemaInstance)
                .withCompressionCodec(CompressionCodecName.SNAPPY)
                .build();
    }

public long flush() throws IOException {
return writer.getDataSize();
    }

public long getPos() throws IOException {
return writer.getDataSize();
    }

public void close() throws IOException {
writer.close();
    }


public void write(T t) throws IOException {
final Tuple2<IntWritable, Text> tuple = (Tuple2<IntWritable, Text>) t;
final List values = new ArrayList();
        GenericRecord record = new GenericData.Record(schemaInstance);
        String inputRecord=tuple.f1.toString();
        Address address = MAPPER.readValue(inputRecord,
                                                    Address.class);
        record.put("id", String.valueOf(address.getId()));
        record.put("customerId", address.getCustomerId());
        record.put("street1", address.getStreet1());
        record.put("street2", address.getStreet2());
        record.put("city", address.getCity());
        record.put("state", address.getState());
        record.put("country", address.getCountry());
        record.put("zipCode", address.getZipCode());

writer.write(record);
    }

public Writer<T> duplicate() {
return new SinkParquetWriter<T>(schema);
    }
}
```

5. 将数据写入 Elasticsearch

Flink 支持各种 connector，这些 connector 作为 Flink 发行版的一部分。在 Flink 发行版中，也有一个适用于 Elasticsearch 的 connector。正如读者所见，向索引中写入文档时，文档 id 与源文档中的 id 相同，而不是使用随机生成的文档 id。另外，我们还指定了 bulk size 和 bulk flush interval，以获得 Elasticsearch connector 的最佳吞吐量。

下面的代码片段是针对 address 数据对象的 Elasticsearch Sink。针对 contact 数据对象的代码与之类似。诸如 esHost、esPort 之类属性可以通过命令行参数传入。读者可以看到，

非卖品，仅供非商业用途或交流学习使用

向索引中写入文档时，文档 id 与源文档中的 id 相同，而不是使用随机生成的文档 id。另外，我们也指定了 bulk size 和 bulk flush interval，以获取 Elasticsearch connector 的最佳吞吐量。

```
//Elasticsearch Sink
Map<String, String> config = Maps.newHashMap();
config.put("bulk.flush.max.actions", "1000");
config.put("bulk.flush.interval.ms", "250");
config.put("cluster.name", "elasticsearch");

List<InetSocketAddress> transports = new ArrayList<InetSocketAddress>();
transports.add(new
InetSocketAddress(InetAddress.getByName(parameterTool.getRequired("esHost")
),
Integer.parseInt(parameterTool.getRequired("esPort"))
                                        ));

messageStream.addSink(new ElasticsearchSink<Tuple2<IntWritable,
Text>>(config, transports,
new ElasticsearchSinkFunction<Tuple2<IntWritable,Text>>() {
public IndexRequest createIndexRequest(Tuple2<IntWritable, Text> element) {
return Requests.indexRequest()
                        .index("address")
                        .type("address")
                        .id(element.f0.toString())
                        .source(((Text) element.getField(1)).toString());
            }

public void process(Tuple2<IntWritable, Text> intWritableTextTuple2,
RuntimeContext runtimeContext, RequestIndexer requestIndexer) {
requestIndexer.add(createIndexRequest(intWritableTextTuple2));
            }
        }));
```

命令行参数

为了运行 Flink ingestor，需要将代码编译成一个独立可执行的 JAR 文件，在工程的 chapter10 目录中运行下面的命令。

```
mvn install
```

这将生成两个 JAR 文件，以便在 Flink 中进行部署，一个 JAR 文件是 speed-address-flink-ingestor-SNAPSHOT-1.0.jar，另一个是 speed-contacts-flink-ingestor-SNAPSHOT-1.0.jar，它们分别存在于各自对应的工程目录中。

一旦生成 JAR 文件，需要将它部署在 Flink 中，并且按下面的方式传递命令行参数，以便代码拥有执行所需的所有细节信息。当在 Flink 容器中运行 Flink 流水线时，遵循同样的方法。下面分别是处理 address 及 contact 数据所需的参数：

```
--topic address --bootstrap.servers <kafka-ip-address>:9092 --
zookeeper.connect <zk-ip-address>:2181 --group.id 1 --hdfsPath
hdfs://<hadoop-namenode-ip-address>:9000/datalake/raw/address --esHost <es-
```

非卖品，仅供非商业用途或交流学习使用


```
ip-address>  --esPort 9300 --auto.offset.reset earliest

--topic contacts --bootstrap.servers <kafka-ip-address>:9092 --
zookeeper.connect <zk-ip-address>:2181 --group.id 1 --hdfsPath
hdfs://<hadoop-namenode-ip-address>:9000/datalake/raw/contact --esHost <es-
ip-address>  --esPort 9300 --auto.offset.reset earliest
```

下面将介绍如何以在 Flink 中以 Flink job 形式部署这些 JAR 包。

6. Flink 部署

Flink 流水线部署是一个简单的过程。然而，它可能会面临多个库之间版本不匹配的挑战。根据经验，最好是重用 Flink 库作为发行版的一部分，然后添加其他独立的库。

当 JAR 包作为 Flink job 被提交时，Flink 会在其 classpath 中搜寻依赖的包，如果某个被依赖的库找不到时，job 自然也就无法运行。

为了提供 Flink classpath 中所有的依赖包，最好在将 JAR 包作为 job 提交执行之前，将所需的包复制至 ${FLINK_HOME } /lib 目录中。这考虑了所有和 Hadoop、Avro 相关的依赖。

然后，还有少量的库需要被额外添加到 Flink classpath 中去。在此处的例子中，需要使用下面的命令来复制 flink-hadoop-compatibility 包（假设 flink-hadoop-compatibility 包已经根据源码构建并安装了）：

```
cp ~/.m2/repository/org/apache/flink/flink-hadoop-
compatibility_2.10/1.3.0/flink-hadoop-compatibility_2.10-1.3.0.jar
${FLINK_HOME}/lib
```

对于 Flink，如果想获取其他依赖项，也建议复制某些 JAR 包（speed-address-flink-ingestor-1.0-SNAPSHOT.jar 或 speed-contacts-flinkingestor-1.0-SNAPSHOT.jar） 至 ${FLINK_HOME}/lib 目录中。如果读者已经在本地机器构建了 JAR 包，可使用下面的命令：

```
scp <source-file-jar> <centos_user>@<ip-of-vm>:~/
```

另外，也必须确保 Flink 流水线执行所依赖的所有系统已运行。这些系统包括 HDFS、Kafka、ZooKeeper、Elasticsearch。

一旦这些先决条件就位，就可以构建示例并将它们作为 Flink 流水线部署到 Flink 容器中，请遵循以下步骤：

1）使用下面的命令启动 Flink 本地容器：

```
${FLINK_HOME}/bin/start-local.sh
```

2）登录至 Flink（URL: http://ip-of-vm:8081），使用前面讨论过的相同的命令行参数来提交 job，同时上传下面这些 Flink ingestor JAR 包。请确保为已提交的 job 提供对应的命令行参数。与之前例子不同之处仅为 topic 名称及 HDFS 路径。下面是完整的参数集，可供使用者参考：

非卖品，仅供非商业用途或交流学习使用


```
--topic address --bootstrap.servers <kafka-ip-address>:9092 --
zookeeper.connect <zk-ip-address>:2181 --group.id 1 --hdfsPath
hdfs://<hadoop-namenode-ip-address>:9000/datalake/raw/address --
esHost <es-ip-address>  --esPort 9300 --auto.offset.reset earliest
```

图 10-34 展示了 Flink job 部署。

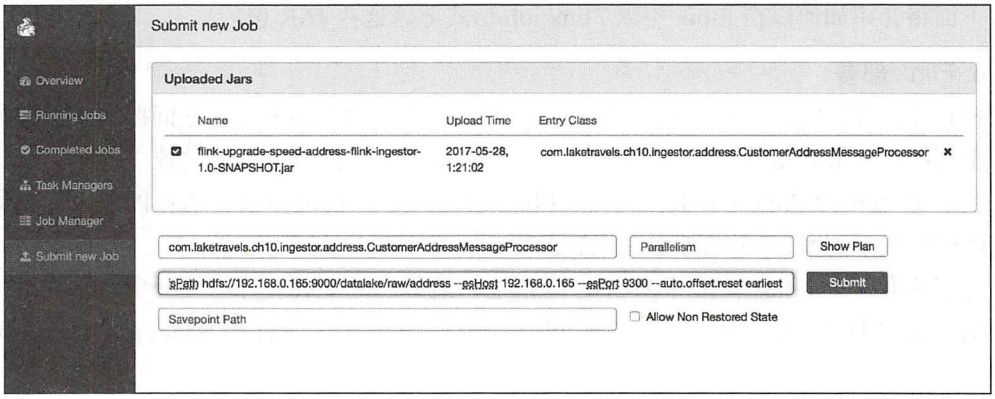


图 10-34 Flink job 部署

为了检验 Elasticsearch 是否收到了来自 Kafka 的命令，可执行下面的命令，之后就能看到 Elasticsearch 中增长的文档数：

```
curl -XGET <ip-of-VM>:9200/_cat/indices?v
```

如果没有显示文档数，则执行前面详细讨论过的 Flume 命令来生产消息并发布至 Kafka 队列中。这将最终触发 Flink 的消息处理操作。

3）一旦 job 提交成功，Flink UI 会跳转到 job 状态及管理页面，如图 10-35 所示。

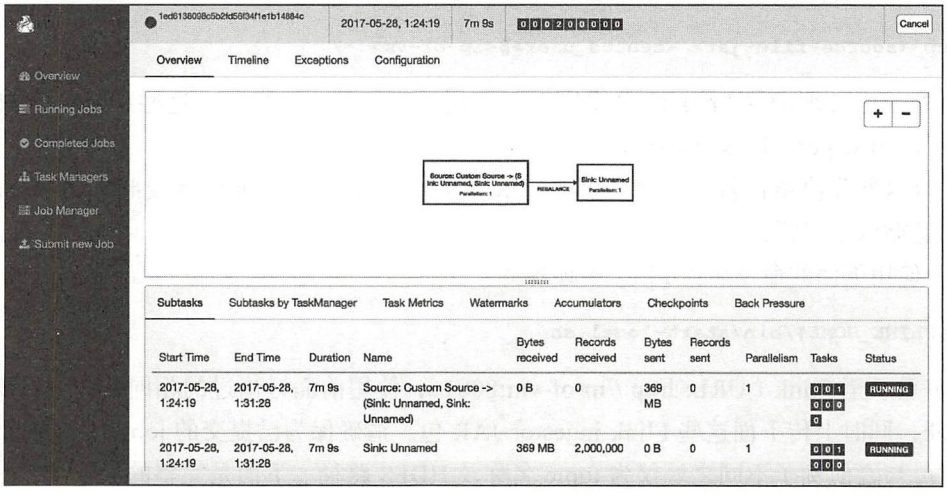


图 10-35 job 状态页

非卖品，仅供非商业用途或交流学习使用

4）类似地，address 对应的 job 也可并行部署到 Flink 流水线中去。读者可以看到每个 topic 对应的事件被消费，事件携带的数据被同时写入 HDFS 和 Elasticsearch 中去。

7. Parquet 数据的可视化（Hive Table 形式）

一旦 Parquet 数据文件被摄取出来，可以创建 Hive 外部表，然后利用 Hive 表轻松查看文件中的数据，下面可以看到相应的脚本。该脚本可以通过 HueUI（Hive Query Builder）来执行，前面几章中已经详细介绍过如何在 Hue 中执行脚本：

```
SET mapred.input.dir.recursive=true;

CREATE EXTERNAL TABLE customer(id int, first_name string, last_name string,
dob BIGINT)
STORED AS PARQUET LOCATION '/datalake/raw/customer';

CREATE EXTERNAL TABLE address(  id string,
                                customerId string,
                                street1 string,
                                street2 string,
                                city string,
                                state string,
                                country string,
                                zipCode string
                                )
STORED AS PARQUET LOCATION '/datalake/raw/address';

CREATE EXTERNAL TABLE contact(id string, cell string, phone string, email
string)
STORED AS PARQUET LOCATION '/datalake/raw/contact';
```

图 10-36 显示了上面脚本创建的表，这里的数据源自 Parquet 文件。

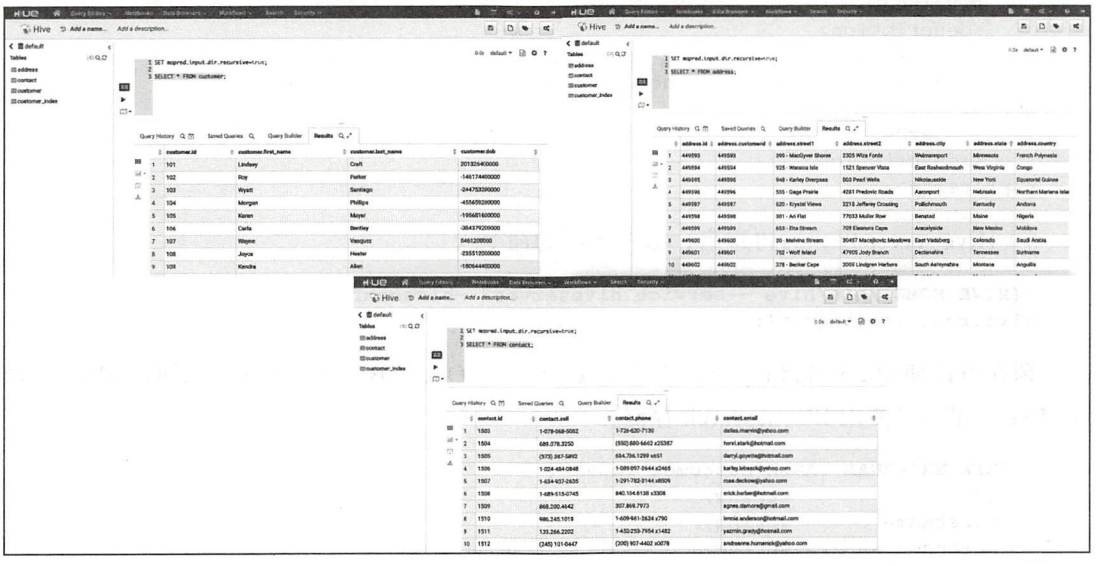


图 10-36　支持 Parquet 文件的 Hive 表

非卖品，仅供非商业用途或交流学习使用

8. 索引 Hive 中的数据

Hadoop 中有全部的 customer 数据，现在可以通过 Hive 表以可视化方式展示已经加载到 Hadoop 中的数据。通过 Flink 流水线，同时向 Elasticsearch、Hadoop 中写入了 address 及 customer 数据。此时也可以通过 Sqoop job 来访问 Hadoop 中的 customer profile 数据，但是 Elasticsearch 中没有此类数据。

可将 Hive 中数据导出并索引至 Elasticsearch 中。使用 ES-Hadoop 框架能轻松实现该目标，ES-Hadoop 是 Elastic Stack 的一个组件。

为了让 ES-Hadoop 框架与 Hive 协同工作，需要进行一些快速的设置与配置，主要步骤如下：

1）使用下面的命令，从指定位置下载 ES-Hadoop 二进制安装包：

```
wget
http://download.elastic.co/hadoop/elasticsearch-hadoop-5.4.0.zip
```

2）切换至某用户目录，使用下面的命令解压安装包：

```
unzip ${DOWNLOAD_DIR}/elasticsearch-hadoop-5.4.0.zip
```

3）通过 ${ES_HADOOP_HOME} 来引用解压目录。

4）对 ${HIVE_HOME}/conf/hive-site.xml 文件按下面的方式进行配置（配置文件中已经有 property 占位符了）。用完整路径来替换 ${ES_HADOOP_HOME}：

```
<property>
    <name>hive.reloadable.aux.jars.path</name>
    <value>
${ES_HADOOP_HOME}/dist/elasticsearch-hadoop-hive-5.4.0.jar
    </value>
    <description>
       Jars can be renewed by executing reload command. And these
       jars can be used as the auxiliary classes like creating a
UDF or
       SerDe.
    </description>
</property>
```

5）Hive server 停止后再重启，参考下面的命令：

```
${HIVE_HOME}/bin/hive --service hiveserver2 -hiveconf
hive.root.logging=info
```

现在可以使用 ES-Hadoop 了，在 Hue UI（Hive Query Builder）中执行下面的脚本来创建 Hive 外部表（使用 Elasticsearch 存储）：

```
CREATE EXTERNAL TABLE customer_index (
   id              string,
   firstName       string,
   lastname        string,
   dob             bigint
   )
```

非卖品，仅供非商业用途或交流学习使用


```
STORED BY 'org.elasticsearch.hadoop.hive.EsStorageHandler'
TBLPROPERTIES( 'es.resource' = 'customer/customer',
               'es.url' = '<es-server-ip-addr>:9200',
'es.mapping.id' = 'id'
             );
```

一旦外部表创建完毕，运行下面的查询来加载外部表灌入数据。查询执行后会在 Elasticsearch server 中为前面定义的每一个表创建一个 customer 索引和一个 customer _type：

```
SET mapred.input.dir.recursive=true;

INSERT OVERWRITE TABLE customer_index select id, first_name, last_name, dob
from customer;
```

到现在为止，已经通过 ES-Hadoop 在 Elasticsearch 存储中创建了一个 Hive 视图。与 customer 索引对应的映射会根据数据自动推导出来，因为 Elasticsearch 支持无模式索引（schema-less indexing）：

```
GET customer/_count
GET _mapping/customer
```

图 10-37 展示了 customer 索引文档计数与类型映射。

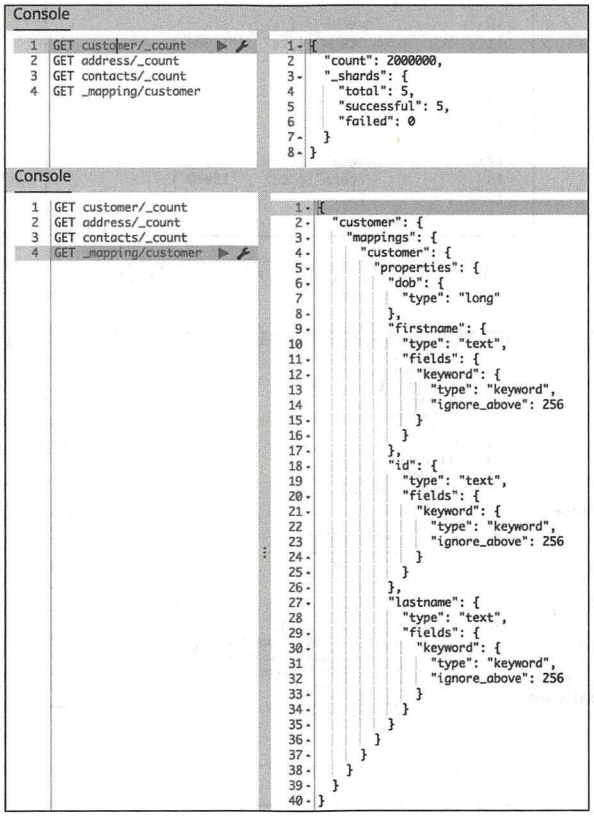


图 10-37　customer 索引文档计数与类型映射

非卖品，仅供非商业用途或交流学习使用

9. 从 Elasticsearch 中查询数据（costomer、address、contact）

现在，Elasticsearch 拥有所有的数据了，其中包括从 Hadoop 层合并来的 customer profile 数据。此时可以发挥 Lambda 架构的优势来查询这些数据，如下所示：

```
POST customer/_search
{
  "query" : {
    "term":{"id" : "18000"}
  }
}
```

图 10-38 中的 Query 执行完毕以后，会在 Chrome 的 Sense 插件中显示查询结果，如图 10-39 所示。

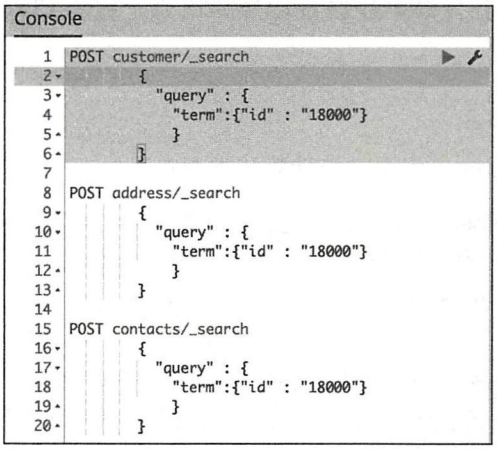


图 10-38 用于检索已索引记录的 Query

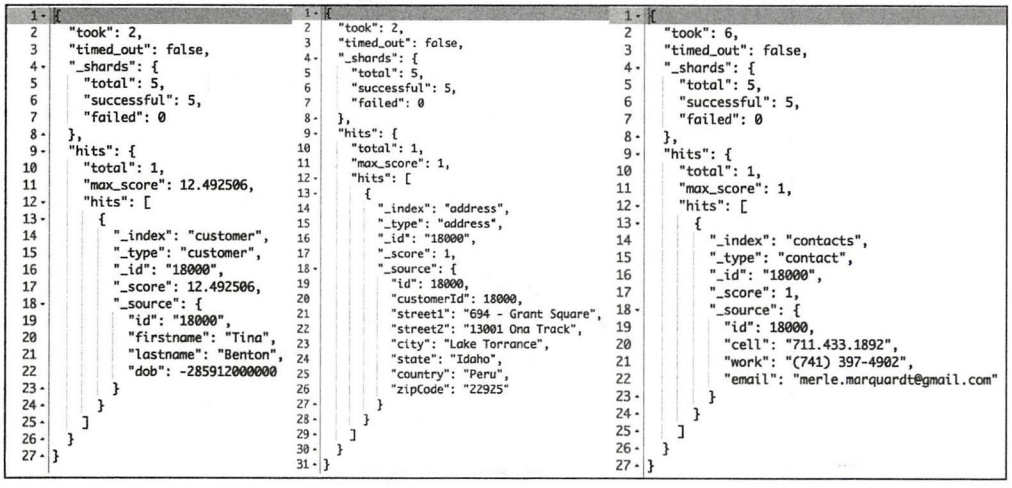


图 10-39 Elasticsearch 查询响应

非卖品，仅供非商业用途或交流学习使用

如图 10-39 所示，我们对 id 字段进行了搜索。然后，从性能角度考量，为实现轻量级的、真正的实时访问，按 id 检索文档是更好的策略。

尽管这里可以通过 _id（每个文档在 Elasticsearch 内部的 id）来检索文档，然而这里的目标是向读者演示 Elasticsearch 的查询能力。如果在索引期间没有显式指定 id 的映射，Elasticsearch 会为文档自动生成一个内部 id。这将制约用户使用 id 来检索文档。图 10-40 显示了 SCV 用例中的 Elasticsearch 组件。

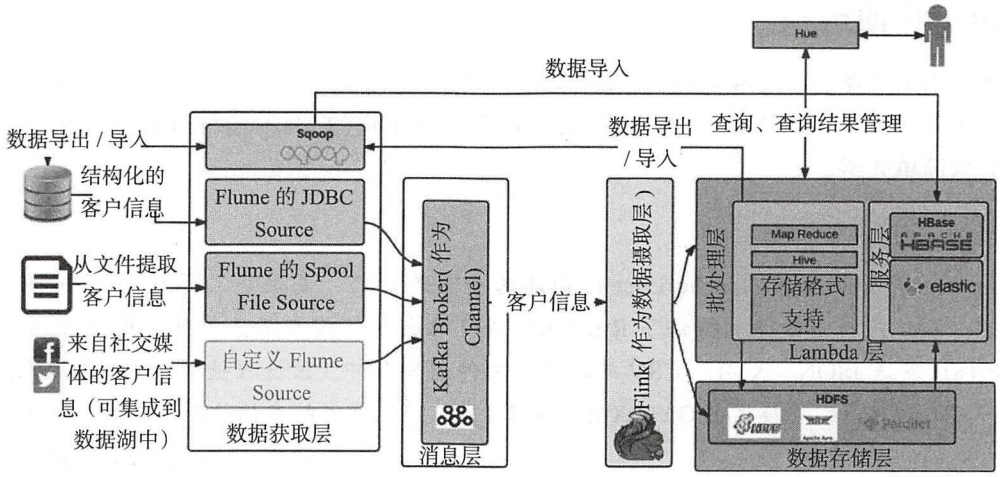


图 10-40　SCV 用例中的 Elasticsearch 组件

10.18　适合使用 Elasticsearch 的场景

以下场景中适合使用 Elasticsearch：

- 需要执行大量基于文本的搜索。
- 对海量非关系数据进行分析。例如，分析各种各样的应用程序日志，这些日志是关系数据，需要基于文本搜索来挖掘其中的语义。
- 执行轻量级查询，尤其是像处理自动完成（auto-completion）之类的基于文本搜索的查找时。
- 用户想寻找一种无模式（schema-less）数据存储时（很显然，Elasticsearch 很适合存储 JSON 格式的文档）。

10.19　不适合使用 Elasticsearch 的场景

- 事务处理密集型。
- 需要在数据存储层之上执行高强度的计算密集型任务。

仅供非商业用途或交流学习使用

- 寄望于 Elasticsearch 作为主存储系统。如果用 Elasticsearch 作为主存储系统，每次插入数据，它需要重建索引，这比较耗时，而且在数据插入或更新时，受影响的索引不可用。
- 用户寻找支持 ACID 事务处理的数据存储系统时。
- 用户寻找持久化数据存储时。

10.20 其他选项

虽然本书选择 Elastic Stack 作为索引数据存储系统，但尽管该选项非常棒，市面上依然有一些不错的同类产品值得考虑。Apache Solr 是一款不错的 Elasticsearch 竞争产品，这里将对它做简单介绍。

Apache Solr

Apache Solr 是基于 Apache Lucene 构建的一款流行的、高性能、开放源码的企业搜索平台。Solr 是一个独立的提供了 REST-like API 的企业级搜索服务软件。用户可以通过 HTTP 协议发送 JSON、XML、CSV 或二进制文件到 Solr 中（该过程被称为索引）。可通过 HTTP GET 对 Solr，并接收 JSON、XML、CSV 或二进制格式的查询结果。

——http://lucene.apache.org/solr/

Solr 的特征列表可参考 http://lucene.apache.org/solr/features.html。对于实现数据湖来说，Solr 是一个非常不错的选择。以下是 Solr 的一些特性：

- 先进的，经过优化的全文搜索功能：利用了 Lucene 先进的搜索功能。
- 处理高流量的能力。
- 基于标准的开放接口：支持 XML、JSON、HTTP 等标准，由于这些标准（或协议），使得应用程序代码编码和维护变得非常容易。
- 全面地管理接口：内置响应式管理用户界面。
- 易于监控：通过 Java Management eXtension（JMX）来发布各种性能指标。
- 高度可扩展、容错：内部使用 Apache ZooKeeper，这使系统易于扩展及实现分布式。
- 灵活的适应性配置。
- 近实时索引：通过使用 Lucene 的近实时索引能力来实现该特性。
- 可扩展的插件机制：内置集成很多插件，用户也能轻松创建一个自定义插件。
- 同时支持模式文档或无模式文档。
- 切面搜索（Faceted search）及过滤功能。
- 基于地理位置的搜索能力：内置的基于位置搜索的特性。
- 高度可配置的文本分析：内置支持多语言处理，同时也提供了很多内置的文本分析工具。

仅供非商业用途或交流学习使用

- 可配置的和可扩展的缓存。
- 内置的安全策略：SSL，用户认证与基于角色的授权。
- 丰富、先进的存储选项。
- 丰富的文档解析能力：内置集成了 Apache Tika，这使索引 PDF、Word 等格式文档的内容变得非常容易。

如果选择 Solr 作为技术栈的一个组件，上面的特性需要仔细了解。

10.21　总结

与本书第二部分其他章类似，本章介绍了数据湖的一个功能模块 / 层及其选用的技术框架。我们给出了选择该技术框架的原因，然后深入研究了该技术框架，包括它的架构、系统组件等。

然后解释了该技术框架的一些重要方面，深入研究了一个实际的工作案例。之后向读者展示了 SCV 用例视角中的 Elasticsearch。在本章末尾，详细列举了哪些场景适合使用 Elasticsearch 以及哪些场景不适合。最后对 Elasticsearch 的一款竞争产品 Solr 进行了简单介绍。

在数据湖实现中，Elasticsearch 是一门非常重要的技术，它是对技术栈的一个有力补充。在介绍完 Elasticsearch 以后，本书的第二部分将告一段落。截至本章，几乎已经覆盖了数据湖相关的所有技术，同时我们也对这些技术之间的联系有了清楚的了解。本章的 SCV 使用案例部分已经实现了完整的数据湖，在本书的第三部分将结合鲜活的案例对数据湖进行整体介绍，届时将介绍 Elasticsearch 与各功能层级（Lambda 批处理层，Lambda 快速处理层，Lambda 服务层）的关系。

读者通读本章内容以后，应该对 Elasticsearch 有了非常清楚的认识，了解其作为数据索引框架，同时也提供了丰富的分析器来帮助实现高效搜索。读者同时也了解了如何使用 Elasticsearch 来构建数据湖，以及如何利用高效的索引分片和分布机制来实现水平扩展及对外提供一致的查询性能。现在，读者应该能更好地认识为什么 Elasticsearch 能用于处理快速数据流及可用于构建高性能应用程序了。

仅供非商业用途或交流学习使用

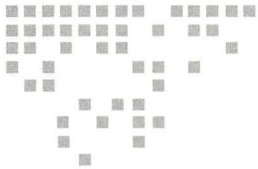
第三部分 Part 3

将所有技术整合在一起

此部分将汇集第一部分和第二部分的技术组件，为读者提供数据湖的整体视图。我们将以简短的方式引入额外的概念和技术，如果有需要，读者可根据企业需求更深入地探索这些方面。深入研究此部分所涵盖的技术超出了本书的范围，但是，我们仍希望读者了解这些附加技术，以及在需要时如何将它们引入数据湖实现。此部分共分两章，如下所示：

- 第 11 章　数据湖组件集成
- 第 12 章　数据湖用例建议

仅供非商业用途或交流学习使用



Chapter 11 第 11 章

数据湖组件集成

恭喜各位读者，一路前行，终于来到了这一章，太棒了。

此时此刻，如果读者已经在阅读各个章节时同时完成了编码练习，其实读者已经不知不觉地基本实现了整个数据湖。

本章将此前的实施数据湖的松散成果整合起来，并提出一些建议和考虑因素，以供读者在自行实施数据湖时能够参考。

本章先会对 SCV 用例的要求和目前取得的成果进行对比，并尝试缩小它们之间的差距。然后将介绍数据湖实施的相关内容，前面的章节并没有详细讨论这方面的内容。

此外，我们还会提出一些建议，以供读者在实施数据湖时采纳。

本书的策略是，让读者在学习前一部分时就几乎可以完成数据湖的实现，但仍然缺乏全局的认识，而在本章之后，将让读者获得对数据湖的全面的认识，并掌握相关内容（本书第三部分）。

11.1 数据湖的学习进程

通过对本书第二部分的学习，读者对数据湖实现已经有比较完整的认识，图 11-1 中展示了目前已经实现的部分。

表 11-1 中简要回顾了各组件的作用。

截至目前，在数据湖中，数据从各种数据源系统流经各个组件并被持久化存储起来。用户还可以通过一些机制来查看数据湖中的数据并进行分析。

仅供非商业用途或交流学习使用

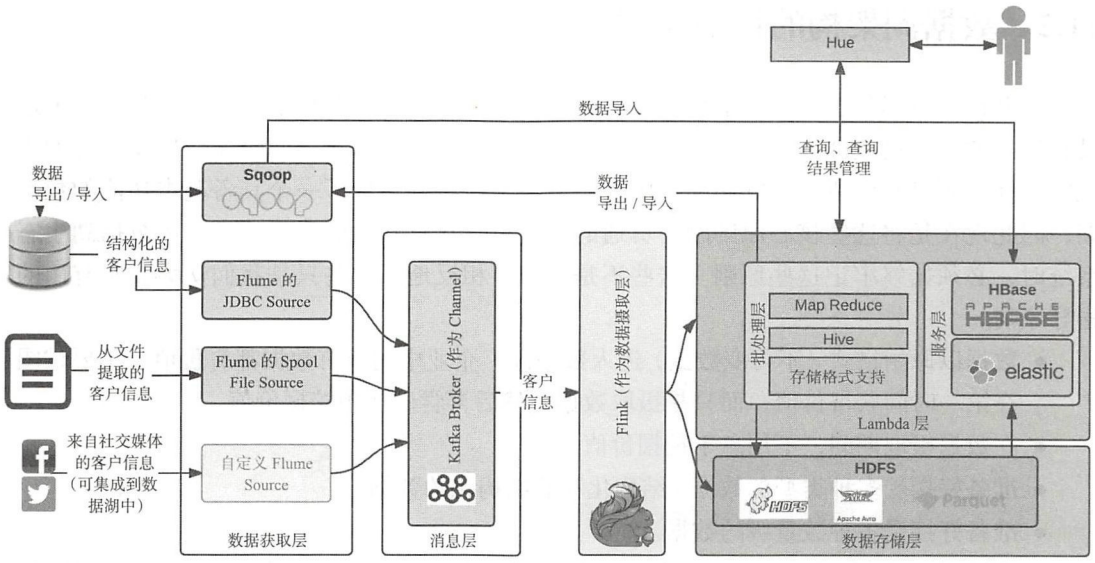


图 11-1　本书目前已实现的数据湖

表 11-1　数据湖中各组件作用

HDFS	分布式文件存储
MapReduce	批量处理引擎
YARN	资源调度管理
HBase	列式键 / 值对 NoSQL 数据库，运行在 HDFS 之上
Hive	查询引擎，提供了类 SQL 的对 HDFS 中数据进行查询的接口
Impala	快速查询引擎，用于执行对 HDFS 中数据的分析查询
Sqoop	数据获取及数据摄取
Flume	数据获取及流式事件数据摄取
Kafka	高度可扩展的分布式消息引擎
Flink	全功能的实时数据处理引擎，同时支持批量计算
Spark	全功能的快速批量数据处理引擎，同时以微批处理的方式支持实时处理
Elasticsearch	基于 Lucene 的快速分布式索引引擎，也可以作为基于文本的 NoSQL 数据存储

这并非数据湖的终点，因为还有很多附加功能可以在数据湖的基础上得以实现，从而让组织能充分利用它。

在接下来的章节中，我们将介绍这些功能，并在实现这些功能的技术选型方面提供建议。

在实现这样一个数据湖时，笔者也会提出自己认为较为重要的考量和建议。这些还只是一些构想，并不具有权威性，更确切地说，是为数据湖的现在和将来而考虑。

在某些场景中，本书将涵盖一些新的技术，但是对技术只会粗略地介绍，因为深入研究其中的每一项技术已超出本章的范畴。

仅供非商业用途或交流学习使用

11.2 数据湖架构的核心原则

本书在前面部分已经提及了在实施数据湖时应该遵循的一些核心原则。但是，本书还没有显式地提及这些准则，因为提前揭示可能会令读者望而却步，同时无法启发读者的主观能动性，而只是停留在数据湖的实施层面。由于已经构建好了一个具备基本功能的数据湖，因此现在是将这些核心原则融会贯通的好时机，在将新功能和新技术引入数据湖生态系统时，必须始终牢记这些原则。这些不是权威，相反地，这些只是我们认为非常有用的指导原则。

- 数据以原始格式（不可变数据）进入数据湖。企业中的所有数据都有价值。不要试图在第一时间获得价值，而只是摄取数据，然后再尝试从中挖掘价值。
- 在数据摄取期间，不用急于挖掘价值。
- 准备好接受多种类型的数据（结构化和非结构化的数据）。
- 准备好接受各种数量级的数据。
- 不要限制数据存储，这样用户可以在数据湖中查询数据。根据需求引入各种技术，来进行各种分析。
- 为企业应用程序访问数据提供简单的方法。最初这些数据没有太大意义，但一段时间后，这些数据便能够与数据湖中的其他数据元素协作，并为企业带来价值。
- 存储时无须对数据进行标准化处理。
- 能够快速、简单且低成本（高度可伸缩）地添加数据源。
- 能够根据消费需求提供各种格式的企业数据给应用程序使用。
- 能够通过数据聚合和大规模数据处理来支持数据智能需求。
- 能够在系统运行或空闲时清理数据。
- 能够支持各种安全机制。
- 由于需要提供重要的企业数据，因而必须高度可用。
- 不要强制改变进入数据的格式，而是以传入数据的格式来接收数据。
- 尽可能减少数据量和网络 / 带宽需求。通过使用多种压缩方法来实现这一点。

11.3 企业数据湖面临的挑战

了解构建和管理企业数据湖时可能面临的挑战对读者不无裨益。这里只讨论数据湖面临的各项挑战，而技术选型、业务应用以及如何寻求企业高管支持等内容不在这里讨论。其中部分挑战以及我们建议采取的措施如下：

挑战 #1 ：如果读者正在使用免费的开源技术来构建数据湖，那么跟上这些技术发展的步伐可能是一项颇具挑战性和艰巨性的任务。

仅供非商业用途或交流学习使用

措施 #1：如果业务方积极采用数据湖，可考虑使用如 Cloudera、Hortonworks 等商业产品。

挑战 #2：如果为达到预期的结构，数据湖使用了大量的技术，那么跟上这些技术及其依赖的其他技术的发展速度可能再次成为挑战。

措施 #2：与前面的措施类似，采用具有商业支持的技术平台可以作为缓解这类的问题的方案。

挑战 #3：曾经，在大数据领域招聘技术熟练的人员极具挑战性。不过现在情况已经有所改善，然而，寻找真正熟练掌握实际操作技术的人员，来维护和管理数据湖中多样化的技术仍然困难重重。

措施 #3：可以考虑与企业建立一个卓越中心（Centre of Excellence，CoE），并制订计划让该团队及时获得更多资源，不断成长。

挑战 #4：构成数据湖的各种组件持续独立演变，数据湖内部、外部的集成变得非常具有挑战性。

措施 #4：可以考虑引入一个中间层来应对数据湖内部和外部应用程序集成的挑战。该层隐藏了详细信息，并提供了统一的方法让数据流入数据湖。

挑战 #5：由于企业应用程序的多样化，将数据集成到数据湖将是一大挑战。如果企业中的应用程序是内部应用程序和供应商应用程序的结合体，那么这个问题会变得更加棘手，因为这些应用程序是用不同语言编写的，也具有不同的数据摄取能力。

措施 #5：如果应用程序能够以非常简单的方式获取数据，比如，对于 Java 应用程序（特别是内部构建的），则在模型中添加一个 Java 注解，这样就可以自动将数据摄取到数据湖中，那么将应用程序数据摄取到数据湖就变得相对容易。

挑战 #6：对常见的数据存储来说，数据授权（安全性和隐私性）将是一个大问题。

措施 #6：为数据和所有数据湖组件提供足够的安全性，严格控制各种用户的权限。这样可能会限制数据分析，这取决于数据湖中的数据以及访问这些数据的人员。

挑战 #7：即使数据源没有问题，通常情况下数据质量仍可能存在问题。数据治理也是一个挑战。需要领域知识加持才能确保进入数据湖中的数据质量最高。

措施 #7：可以加入一个流程，在数据获取过程中嵌入适当的数据治理框架，并及时检查数据质量。在一段时间内将对数据质量的把控上升到组织文化的高度。

挑战 #8：通常，大型组织缺乏跨所有部门的信息 / 业务架构。正因为如此，基于原始数据构建适当的数据模型非常困难。这将给 IT 部门带来额外开销，因为需要创建不断变化的数据模型，以供业务用户进行各种分析。

措施 #8：使用数据湖开始构建信息 / 业务架构，并使用新约定的数据模型在数据湖中进行数据建模。

挑战 #9：很难实现完全的自动化，这对系统维护构成了挑战。可能会导致高昂的维护成本，但一般不会超过数据湖所带来的收益。

仅供非商业用途或交流学习使用

措施 #9：先致力于实现小范围自动化并在一段时间内不断推动其全面自动化。

11.4 企业对数据湖的期望

构建和管理数据湖的成本不菲。因此，各方对数据湖都抱有相当苛刻的期望，而且这些期望不尽相同。按照企业内人员角色，这些期望可以划分成两类。

业务用户的期望：

- 能一直在高质量的正确数据上进行分析。
- 可以简单地进行数据治理工作。
- 能够细粒度地控制数据的可见性来设置安全措施。具备简单的数据屏蔽功能，在需要进行某些数据转换操作时要授权机制控制用户权限。
- 具有自助服务能力，大部分用户不需要有技术背景就能使用。
- 能够很容易地表示数据世系和可追溯性。
- 能够支持元数据管理。

数据世系（data lineage）指的是数据的生命周期，其中包括数据的源头和随着时间变化的情况。它描述了数据在各个处理过程中的变化情况，提供了对数据分析流水线的可视性，降低了向数据源回溯排查错误的难度。

——维基百科

技术部门的期望：

- 能够简单地从各个应用程序中摄取数据。例如，云应用程序、内部构建的应用程序，即使这些应用程序是采用不同技术开发的。
- 数据湖的管理和维护尽量简化，能有效降低企业成本。
- 能够根据行业需求，易于引入维护数据湖的人员和技能。
- 应该能可伸缩地执行所需的数据处理并智能导出。
- 能够满足机器学习以及其他数据算法的需求。
- 必须具备故障安全性和可靠性机制，以便于故障恢复和保障可用性而不会中断业务。

11.5 数据湖的其他用途

数据湖的架构庞大而昂贵（在生产环境中，理想情况下我们使用高端硬件而不是普通商业硬件），它还有其他的一些潜在用途。高端基础设施面临的主要挑战是提高有效利用率。虽然可能需要高端基础设施来解决棘手问题，但很可能利用率不高。这里需要考虑采用一些机制来提高这些设施的利用率。

仅供非商业用途或交流学习使用

最实际的方法就是采用 Hadoop 架构的多租户机制。仔细了解 Hadoop 或任何存储系统，在存储层执行两项基本操作：一个是读取数据，另一个是为了数据存储和处理的目的而写入数据。

通过利用整个基础架构中各个组件支持的安全机制，可以在基础层面上实现上述目标，以便安全团队可以在共享基础架构上提供所需的业务隔离。必须设置安全准入原则来让各个团队和用户使用。这也有助于防止组织内出现数据湖孤岛问题。

这些功能还可以吸引组织的应用程序和团队将数据摄取到数据湖中。这里包括了两个基本方面：

- 数据存储（HDFS，分布式文件系统）：数据湖的核心组件是 Apache Hadoop，它能够以各种格式存储海量数据。数据湖是存储大量应用数据（例如审计和日志等数据）的潜在备选方案，并且应用程序可以利用它以非常便宜且易于访问的方式存储大量数据。它也可以用作企业档案管理的解决方案（一个具有简易数据检索能力的集中解决方案）。
- 数据处理（MapReduce 编程模型）：Hadoop 具有处理海量数据的潜力。数据湖的各种内部和外部用例都可以利用这项能力。对于许多用例来说，海量数据处理是必需的。使用 Hadoop 的 MapReduce，可以将这些海量数据处理集成到数据湖的基础架构中，然后将处理过的数据再写回应用程序的数据存储中，以满足更多需求。

11.6　了解更多关于数据存储的信息

数据存储是数据湖最关键的部分之一。Apache Hadoop（HDFS）是数据湖的核心存储组件。图 11-2 非常清晰地展示了这一方面，显示了数据湖中的批量和流式数据存储组件，以及 Hadoop 生态系统中其他与数据存储各方面相关的技术。

接下来将了解数据存储领域中的一些重要概念。我们也将集中地关注批处理和快速处理数据，以及这些数据如何存储在数据湖中，同时还将了解与这些数据类型相关的细节。

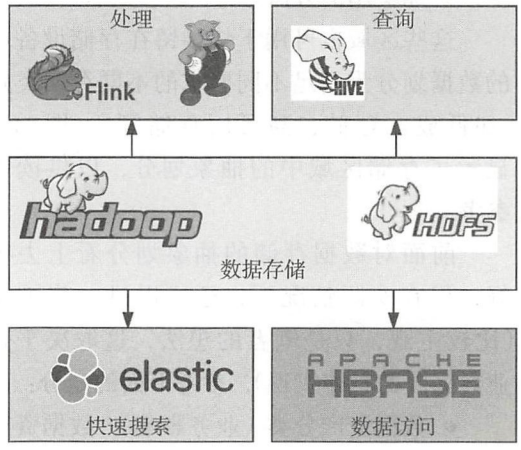


图 11-2　使用 Apache Hadoop（HDFS）作为数据存储系统

11.6.1　数据存储区域

尽管存储系统中的数据不需要遵循某种特定的模式，但对于一个使用数据湖的组织来说，最好有一些明确的方向和原则来

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

说明如何将数据接入数据湖。这些只是在存储系统中构造数据时可以参考的一些建议。

数据可以按多种方式来组织，这取决于用户的组织架构。从较高程度抽象的层面来看，数据可以做如下划分：

- 主数据区（master data）：数据湖中存储主数据的区域。通常来说，这些数据可供整个企业使用，并且可以随着时间的推移而成为企业中唯一的数据来源。主数据对于各种分析都非常重要，因为交易数据会与主数据合并，向数据分析师或业务人员提供有意义的数据可视化。
- 原始数据区（raw data）：来自各个数据源系统的原始数据流入并保存在该存储区域。该区域完全可控，只有数据湖的核心技术人员和数据科学家才能访问这些数据。此处的数据将包含企业中几乎所有的内容，并且包含与组织内某些部门密切相关的敏感信息。
- 企业高级数据区（enhanced enterprise data）：原始数据在该区域中进行建模，产生在组织中通用的数据并保存起来。这个区域也被很好地控制，但是需要依据从原始数据中创建出来的模型，这些模型可以被企业中的各种应用程序使用。这些模型来自为组织创建的信息模型（业务架构）。业务和 IT 部门都能很好地理解该模型。在创建这些模型时，原始数据中仍然有很多属性没有被正确地选择或充分地理解，因此不能包含在模型中。
- 应用程序数据区（application/curated specific data）：更常用的或所谓的精选模型存储于该区域中。通常数据湖是为企业提供的，因此不鼓励应用程序特定的模型，但是为业务应用程序定制模型的情况又屡见不鲜。因此，数据湖中的一些业务应用程序会摄取数据，构造程序特定模型，也能获得数据湖的功能和处理能力。当一个企业启动数据湖建设，并为业务应用程序提供策略以将数据注入数据湖时，这是一种更加实用的方法。

这些区域也可用于将驻留在存储设备中的数据划分为满足不同要求的不同存储类别（如低成本存储、高可用存储等）。图 11-3 显示了存储区域中的抽象划分，以供读者参考。

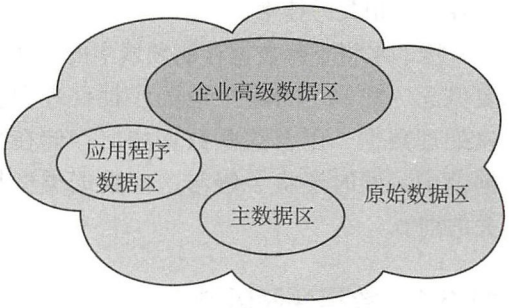


图 11-3 数据存储层中的存储区域

前面对数据存储的抽象划分看上去很好，但在实际情况下，还可以进一步细化（比较主观，只是笔者的想法，这取决于企业数据湖的具体实现），参考下面的划分：

- 数据资产分类（业务影响）：数据资产或业务资产的安全级别划分。
  - 高

非卖品，仅供非商业用途或交流学习使用

  - 中
  - 低
- 数据存档策略：根据数据湖中的数据保存周期。
  - 短
  - 中
  - 长
- 业务领域：数据湖中数据的业务领域。
  - 消费者
  - 销售人员
- 数据源：数据湖中的数据来源。
- 内部或外部。

总而言之，这些区域被保存在存储系统中，以便为数据科学家提供良好的可见性，从而推断出有用的见解，然后根据它们做出有意义的商业决策。

在下一节中，我们将深入研究批处理和高速数据以及一些与之相关的需要注意的细节。

11.6.2　数据模式和模型

原始数据在被摄取时，其格式可能并不符合预期，因为它很可能是按照原始格式存储于数据湖中。从存储系统中查询数据时，返回结果必须存储在符合目标模式的结构中。为了将数据从其原始数据模式转换 / 重构到目标数据模式，需要进行处理。在我们的例子中，最终批处理视图在创建时将具有一个定义明确的模式，在此之上构建视图并提供给用户。

11.6.3　存储选项

前面的章节中提到过多种存储数据的方法。为了能妥善地使用这些存储选项，最好先在一定程度上理解它们。

Hadoop 运行用户使用基本的 HDFS 命令来加载原始数据，并且可以将这些数据写入 Hive 或 Impala 视图进行可视化展示。前面的章节中也提到过这一点。除此之外，我们还使用某些序列化反序列化（SerDe）适配器来告诉视图如何处理数据。

在处理包含最天然形式的数据的 RAW 文件时，文件和数据将被加载到 Hadoop 中进行进一步处理。如果这些 RAW 文件中的数据采用标准格式（如 CSV 文件或 TAB 分隔文件），则通过简单地创建视图就可以很容易地查看这些数据。但是，在这些 RAW 文件中可能会使用非标准格式的数据，这些文件需要进一步处理，因此需要额外的存储格式定义。Hadoop 层内置支持的格式包括纯文本文件（CSV / TAB 分隔文件）、序列文件、Avro 文件和 Parquet 文件等主流文件格式。

格式的选择是由存储数据的目的来决定的。目的和选项的可能组合如表 11-2 所示。

非卖品，仅供非商业用途或交流学习使用

表 11-2 根据不同目的选择文件格式

文本格式	序列文件格式	Avro 格式	Parquet 格式
文本格式数据加载简便，代价是数据压缩率低，但查询开销小	序列文件通常用于打包小文件，用于在多个 MapReduce 作业之间传输数据	Avro 格式是一种二进制格式，它提供基于模式的数据存储并支持块压缩，增强了 I/O 性能，以实现更快、更高效的查询	Parquet 格式是另一种存储数据列的二进制存储格式，通常用于特定列上的查询

HDFS 一般支持这些格式，但根据数据存储位置的不同，存储模式也会有所不同。为了更好地理解这种表述，请读者思考本书中提到的两种 NoSQL 数据存储，即 HBase 和 Elasticsearch。虽然都属于 NoSQL 数据存储，但它们采用了不同的方式来存储和处理数据。HBase 是在 HDFS 上运行的本地非索引数据存储，并以列族形式存储数据，而 Elasticsearch 是一个索引数据存储，以 JSON 文档形式存储数据，为提供高效查询，需要实现一种直接的存储机制。

Apache HCatalog（Hive Metastore）

如果读者的数据湖中使用了 Apache Hive 这项技术（其中集成了 HCatalog），将能非常方便地处理各类技术生态系统产生的多种格式的数据。

HCatalog 是一张数据表，也是 Hadoop 的存储管理层，它使得用户可以使用数据处理工具（如 Pig、MapReduce）更轻松地在网络上读取和写入数据。HCatalog 的表抽象地向用户展示了 Hadoop 分布式文件系统（HDFS）中的数据关系视图，并确保用户不必担心数据存储的位置格式，如 RCFile 格式、文本文件、SequenceFiles 或 ORC 文件。

HCatalog 支持以任何格式读取和写入文件，需要编写 SerDe（serializer-deserializer）。默认情况下，HCatalog 支持 RCFile、CSV、JSON、SequenceFile 以及 ORC 等文件格式。要使用自定义格式，用户必须提供 InputFormat、OutputFormat 和 SerDe。

——https://cwiki.apache.org/confluence/display/Hive/HCatalog+UsingHCat

前文中从 HCatalog confluence 页面摘录的内容很好地描述了 HCatalog（也称为 HCat）的确切功能，本书认为不需要再进行扩展，也没有必要在本章中更详细地介绍。

图 11-4（同样取自 HCatalog confluence 页面）简洁明了地描绘了 HCatalog 的工作原理。

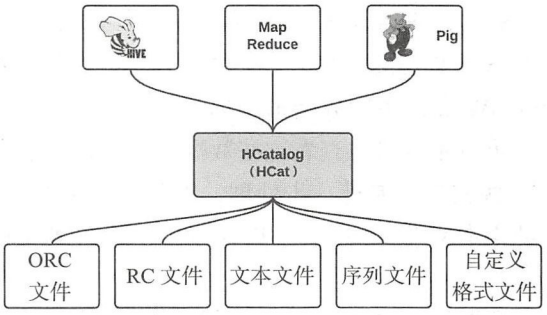


图 11-4 Apache HCatalog 工作原理

非卖品，仅供非商业用途或交流学习使用

11.6.4　压缩方法

在本书前面的部分简要介绍了数据压缩。由于数据压缩是数据湖的一个重要方面，因此在这里对压缩方法进行更为详细的介绍。

在存储数据时，为了优化存储（减少存储空间）和充分利用网络带宽，通常采用压缩方法。数据湖需要处理海量数据，此时数据压缩非常重要。这一点会让数据湖更具可伸缩性，同时也更为灵活。

在企业中存在的许多场景中，通常，数据湖摄取了多种文本格式（CSV、TSV、XML、JSON 等）数据。这些文件是人类可读的，但占用了大量的存储空间。然而在数据湖中，数据应尽可能为机器可读。使用序列化的压缩技术将这些人类可读的数据压缩成机器可读的，可确保存储所需的空间大大减少。以下是一些众所周知的常用压缩格式（称为编解码器，允许数据压缩 / 序列化和解压缩 / 反序列化）：

- Gzip（扩展名为 .gz）：即 GNU Zip（GZip），一种众所周知的压缩格式，并且在互联网世界中被广泛使用。可以使用此格式压缩请求和响应，以有效利用网站 / Web 应用程序的带宽。
- Snappy（扩展名为 .snappy）：由 Google 开发的编解码器（以前称作 Zippy），被认为是中等压缩比中性能最好的。对于该格式而言，性能比压缩比更为重要。Snappy 属于最为广泛使用的格式之一，显然是由于其出色的性能。
- LZO（扩展名为 .lzo）：由 GNU 公共许可证（GPL）授权，与 Snappy 非常类似，它具有中等压缩比，而压缩与解压缩性能很高。
- Bzip2（扩展名 .bz2）：或多或少与 GZip 类似，压缩比高于 GZip。但如预期一样，Bzip2 的数据解压缩速度比 GZip 慢。其中一个重要的方面是，它支持数据分割，这在使用 HDFS 作为存储时非常重要。如果数据只是被存储而无须查询（与本书的用例不同），那么这种压缩将是一个不错的选择。

Lempel-Ziv-Oberhumer（LZO）是一种专注于解压缩速度的无损数据压缩算法。

——维基百科

前面的每种压缩算法都有其优点和缺点。如果压缩比比较高（压缩后节约较大空间），则解压缩速度通常很慢，反之，则解压缩速度快。没有两全其美的办法，只能选择一种方法来使用。这取决于进入系统的数据，必须根据具体情况来选择。

当需要选择压缩算法时，以下选项可供参考：

- 将文件分成若干块，然后使用适合的算法进行压缩。由于选择了 Hadoop（HDFS）作为数据持久化机制，因此需要确保这些数据块在压缩后能与预期的 memory slot 适配（在 HDFS 中配置）。
- 选择允许先分割、再压缩的压缩格式。
- 选择允许分割和压缩的容器文件格式，如 Avro 和 Parquet。这些格式可以与其他压

非卖品，仅供非商业用途或交流学习使用

缩格式混合使用，以达到用户用例所要求的速度和压缩比。在上述两种容器文件格式中，Avro 较为成熟且在业界广泛使用。下面对这两种格式进行简要讨论：

- Avro：使用自解释（隐式创建模式）的二进制格式进行数据序列化。当对数据进行序列化时，同时也序列化了一个模式用于描述该数据。因此，很多编程语言可以很容易地读取这些格式的文件。Hadoop 生态系统中的许多技术，如 Spark、Pig 等，都可以很容易地读取这些文件格式并进行处理。Avro 是一种在网络之间发送数据效果良好的压缩格式。如果需要，也可以使用类似 Snappy 这种压缩格式进行更高阶的压缩。
- Parquet：是一种通过 Apache License 授权的、开源的、完全的列式存储格式。它的名字来源于拼花地板的图案。Parquet 具有非常高效的压缩性能和编码模式。与 Avro 相似，许多 Hadoop 生态系统的技术正在积极支持 Parquet，因为它有不少优点，而且正在大力开发中，每个版本都在不断增加新功能。

请读者根据用例的需求自行选择格式。建议使用合适的容器文件格式和压缩算法，在速度和压缩比之间取得平衡。

11.6.5 数据分区

数据分区（data partitioning）是一种应用非常普遍的技术，因为将持久化存储的数据分隔为多个部分能提升查询性能，同时也易于管理和监控（分隔后，数据变得更小且更易于管理）。这种技术曾在 RDBMS 中使用，在 HDFS 中也可以使用。作为文件系统，HDFS 使用文件分区实现数据分区。对于企业级数据湖来说，数据分区是需要认真思考和对待的事情。前文中提到了数据存储的不同区域，本节中将通过引入数据分区来进行扩展。

存储在 HDFS 中的表具有许多行（每行代表一条记录），每行包含多个列（每列对应记录中的一个属性）。可以通过如下两类方式实现分区：

- 水平分区：基于行 / 记录的划分。在这类分区方法中，会将表中的行划分到不同的分区。
- 垂直分区：它根据列值对数据进行分区。这种分区方法更为常见，如果待执行的查询（尤其是参数）已知，使用这种方法的效果将非常好，并能够提升性能。图 11-5 显示了存储在 HDFS(目录结构)

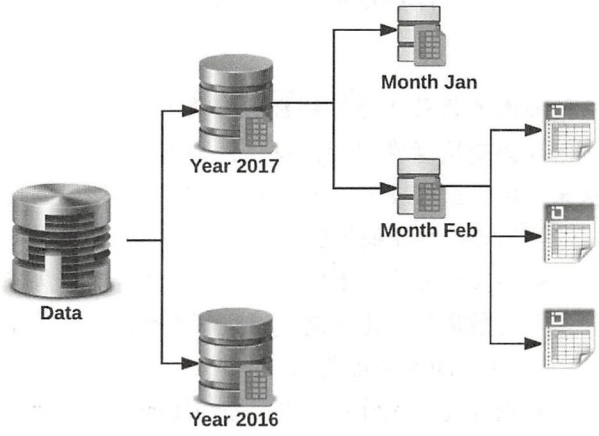


图 11-5 根据列值进行垂直分区（划分为多个层次）

非卖品，仅供非商业用途或交流学习使用

中的示例数据是如何按特定的列值划分成不同的层次的。在第一级目录中，数据按年份进行分区，而在更下层的分区中按照月份进行划分。

11.7　更多关于数据处理的知识

数据处理是数据湖中最重要的功能之一，本书中也不例外，其中的批处理层和快速处理层都与数据处理密切相关。本节将介绍数据湖在数据处理方面一些需要考虑的重要问题。在 Hadoop 1.x 中，主要的数据处理是通过 MapReduce 来完成的。随着 Hadoop 演进到 2.x 版本以及数据摄取的方法越来越多，实时 / 流式计算领域出现了更多的选项，这里将详细介绍两个方面以及各自需要重点考虑的因素。

11.7.1　数据校验和清洗

在数据进入数据湖的持久层（即数据存储层）之前对其进行校验是非常重要的步骤。在数据湖背景中，校验有两方面内涵，具体如下：

- 数据来源：确保来自正确数据源的正确数据被数据湖获取。数据的来源应该是已知的，进入的数据也应该是由数据湖授权摄取的。
- 数据质量：确保对进入数据湖的数据进行了一些初始的属性检查，以此确保进入的数据符合指定的格式。例如，某条数据记录中的一个属性是电子邮箱，则它应该能够通过电子邮箱的格式检验。

校验和恰当的数据清洗在现代数据湖中是一个非常重要的方面，因为在当下，许多数据源系统将大量数据传入数据湖。如果不做校验，数据湖中的数据将会被污染，那么用户要想获取数据并进行各种分析将变得非常困难。在 Hadoop 中创建一些数据区域是一个很好的方法，能够对数据湖中的数据及其质量进行控制。当数据从一个区域移动到另一个区域时，可以根据预先配置好的规则进行校验，以确保数据具有最高的质量，并且某些属性需要符合通用的数据规则（对不同的组织，这些规则不尽相同，但是它们必须是根据业务需求产生的）。

此外，还需要一些规则来确保数据源只能将数据传入数据湖，且每条数据都需要被校验。强制进行数据验证的机制是保存数据模式，以便对任何进入数据湖的数据进行验证，将不合格的数据当作异常进行单独处理。对于这样的用例，使用类似 Avro 的基于模式的存储机制将会非常有用。

数据清洗与数据校验同等重要。在数据校验时，用户就已经做了少量数据清洗的工作。但是，用户需要定期清洗数据湖，以确保问题不会成比例地增长。只有这样，最有价值的、需求最迫切的数据分析才能被及时执行。用户需要一个机制来识别那些随着时间推移而逐渐消失或变得无关紧要的数据，并采取必要的措施将这些数据从数据湖中所谓的生产数据中移除。数据清洗可以通过多种方式来实现，其中一些基本机制包括使用 Hive/Impala 处理

非卖品，仅供非商业用途或交流学习使用

数据，将数据从一张表移动到另一张表进行清洗，然而如果通过查询引擎进行这些操作将会非常耗资源。相反，使用 MapReduce 作业来进行清洗可以获得更好的效果，并且还能集成各种标准库，以进一步简化清洗流程。

以上所述并没有科学依据，因为数据清洗完全取决于企业，并没有一个一刀切的解决方案。但是，这方面非常重要，需要详细考虑和仔细实施。对于批量数据和实时数据，验证和清洗都是需要注意和考虑的重要方面。

11.7.2 机器学习

机器学习是建立在自适应的、持续性的学习系统上的数据科学，用于从数据中抽取有价值的模式或知识。很多基础的算法集能帮助用户从数据中抽取这种知识。这些算法通常可以对输入数据、历史数据进行处理，估算各种事件及其相关性概率，基于输入数据的训练（training），给出推荐或者决策形式的输出，从而提供更好的体验。

机器学习算法大致分为以下几类：

1. 有监督学习算法

有监督学习算法基于已标注数据进行学习，以提供达到预期精度的结果。虽然可以使用神经网络（neural network）模型，但最常见的实现是朴素贝叶斯（Naive Bayes），该模型可以根据训练数据和预定义的类别对数据进行分类。这种模型也可以应用在基于对话中使用的积极和消极词汇而进行情感分析。

2. 无监督学习算法

无监督学习算法能够通过不断分析数据集中的模式和趋势，从而洞察数据中蕴含的意义，但不需要标注任何数据。此类算法主要用于解决数据聚类的问题。该领域中最常见的算法是 K-means，因为该算法可以直接应用于各种数据集，对数据进行迭代式聚类。

当数据驻留在批处理层中时，正好适合运行机器学习算法，可以对数据进行高质量的分析。很多算法还可以用于处理近实时数据，持续进行高精度的数据分析。Apache Mahout 是 Hadoop 领域中非常有用的框架，它具有很多内置算法及范例，能非常便捷地与 Hadoop 进行整合。这些算法很容易应用到欺诈检测、生物识别、面部识别、机器数据的数字信号处理等领域。最流行的例子就是亚马逊的推荐引擎，它根据客户的行为以及过去的购买模式，为客户推荐他们可能感兴趣的购买选项。

11.7.3 调度器 / 工作流

数据湖收集原始格式数据。为了对数据进行有意义的分析，这些原始数据需要经过一定程度的处理。正如前文中介绍的那样，存在某些存储和处理数据的区域确保将原始数据转化为更有利于分析的形式。虽然这些操作可以通过一系列 MapReduce 作业来完成，但主要挑战是如何巧妙地在这些区域之间触发或调度这些作业。另外，由于 Hadoop 领域包含多

非卖品，仅供非商业用途或交流学习使用

种技术和框架，因此各个数据区域可能会需要执行不同类型的任务。下面将介绍如何使用 Oozie 框架来实现该目的，主要通过两个组件（即工作流和协调器）来实现。

11.7.4　Apache Oozie

Oozie 源代码可下载，同时它也需要从源码安装。Oozie 的安装可以按照以下步骤进行。

1. 数据库设置和配置

Oozie 需要一个元数据库来保存 Oozie 的作业、工作流和协调器的信息，因此需要创建一个元数据数据库。与此前管理元数据的方法类似，我们会按照以下步骤在 PostgreSQL 中创建和配置元数据数据库：

1）在 CentOS VM 中的 ssh/ 命令行中进入 PostgreSQL 的提示符界面：

```
psql -U postgres
```

2）运行以下语句创建 Oozie 用户：

```
CREATE ROLE oozie LOGIN ENCRYPTED PASSWORD 'oozie' NOSUPERUSER
INHERIT CREATEDB NOCREATEROLE;
```

3）运行以下语句创建 Oozie 数据库：

```
CREATE DATABASE "oozie" WITH OWNER = oozie
ENCODING = 'UTF8'
TABLESPACE = pg_default
LC_COLLATE = 'en_US.UTF-8'
LC_CTYPE = 'en_US.UTF-8'
CONNECTION LIMIT = -1;
```

现在，已经创建并配置了一个名为 oozie 的元数据库，其用户名为 oozie，用于存储和管理 Oozie 元数据和作业信息。

2. 从源码构建

下一步中，我们将从源码构建 Oozie 并开始使用它。

1）使用以下命令下载 Oozie 源代码的 tar 包：

```
wget http://www-eu.apache.org/dist/oozie/4.3.0/oozie-4.3.0.tar.gz
```

2）切换到用户目录并使用以下命令提取 tar 包的内容：

```
tar -zxvf ${DOWNLOAD_DIR}/oozie-4.3.0.tar.gz
```

3）切换到 Oozie 源代码目录，该目录用变量 ${OOZIE_SRC_HOME} 引用，然后运行以下命令来进行构建：

```
${OOZIE_SRC_HOME}/bin/mkdistro.sh -Phadoop-2 -DskipTests
```

4）构建完成后，可以从 ${OOZIE_SRC_HOME}/distro/target 提取二进制 tar 包到用户

非卖品，仅供非商业用途或交流学习使用

目录中，此目录记为 ${OOZIE_HOME}。切换到 ${OOZIE_HOME} 目录并运行以下命令：

```
tar -zxvf ${OOZIE_SRC_HOME}/distro/target/oozie-4.3.0-distro.tar.gz
```

5）使用以下命令将环境变量 ${OOZIE_HOME} 配置为 Oozie 目录。另外，将这些命令追加到 ~/.bashrc 文件中：

```
export OOZIE_HOME=<directory-containing-oozie-install>
export PATH=$PATH:${OOZIE_HOME}/bin
```

6）提取文件完毕后，按照下列属性在 ${OOZIE_HOME}/conf/ooziesite.xml 文件中配置代理用户和数据库。所有的路径配置必须是完整的路径，因为这里无法解析环境变量。前文提及的环境变量仅供参考：

```
<configuration>
<property>
       <name>oozie.service.ProxyUserService.proxyuser.centos.hosts<
/name>
       <value>*</value>
</property>
<property>
       <name>oozie.service.ProxyUserService.proxyuser.centos.groups
</name>
       <value>*</value>
</property>
<property>
       <name>oozie.service.ProxyUserService.proxyuser.hue.hosts</na
me>
       <value>*</value>
</property>
<property>
       <name>oozie.service.ProxyUserService.proxyuser.hue.groups</n
ame>
       <value>*</value>
</property>
<property>
       <name>oozie.service.ProxyUserService.proxyuser.oozie.hosts</
name>
       <value>*</value>
</property>
<property>
       <name>oozie.service.ProxyUserService.proxyuser.oozie.groups<
/name>
       <value>*</value>
</property>
<property>
       <name>oozie.service.JPAService.jdbc.driver</name>
       <value>org.postgresql.Driver</value>
</property>
<property>
       <name>oozie.service.JPAService.jdbc.url</name>
       <value>jdbc:postgresql://localhost:5432/oozie</value>
</property>
```

```
<property>
        <name>oozie.service.JPAService.jdbc.username</name>
        <value>oozie</value>
</property>
<property>
        <name>oozie.service.JPAService.jdbc.password</name>
        <value>oozie</value>
</property>
<property>
        <name>oozie.service.WorkflowAppService.system.libpath</name>
        <value>/user/centos/share/lib</value>
</property>
<property>
<name>oozie.service.HadoopAccessorService.hadoop.configurations</na
me>
  <value>*=${HADOOP_HOME}/etc/hadoop</value>
</property>
<property>
<name>oozie.service.HadoopAccessorService.action.configurations</na
me>
  <value>*=${HADOOP_HOME}/etc/hadoop</value>
</property>
</configuration>
```

7）在 ${HADOOP_HOME}/etc/hadoop/core-site.xml 文件中加入下列配置：

```
<property>
 <name>hadoop.proxyuser.oozie.hosts</name>
 <value>*</value>
</property>
<property>
 <name>hadoop.proxyuser.oozie.groups</name>
 <value>*</value>
</property>
```

8）运行以下命令重新启动 dfs 服务：

```
stop-dfs.sh && start-dfs.sh
```

9）使用以下命令配置 ${HADOOP_CONF_DIR} 环境变量并将其添加到 ~/.bashrc 文件中：

```
export ${HADOOP_CONF_DIR}=${HADOOP_HOME}/etc/hadoop
```

10）在 ${OOZIE_HOME} 下创建文件夹 libext。将 Hadoop JARS 复制到该文件夹中，这样 Oozie 能使用库文件并执行各种作业。

```
mkdir ${OOZIE_HOME}/libext
```

11）将 ${HADOOP_HOME} 中的所有 Hadoop JARS 复制到 ${OOZIE_HOME}/libext 中。执行以下命令：

```
cp ${HADOOP_HOME}/share/hadoop/common/lib/* ${OOZIE_HOME}/libext
cp ${HADOOP_HOME}/share/hadoop/common/*.jar ${OOZIE_HOME}/libext
```

```
cp ${HADOOP_HOME}/share/hadoop/hdfs/lib/* ${OOZIE_HOME}/libext
cp ${HADOOP_HOME}/share/hadoop/hdfs/*.jar ${OOZIE_HOME}/libext

cp ${HADOOP_HOME}/share/hadoop/mapreduce/lib/* ${OOZIE_HOME}/libext
cp ${HADOOP_HOME}/share/hadoop/mapreduce/*.jar ${OOZIE_HOME}/libext
```

12）使用以下命令在 HDFS 中配置 Oozie 的共享库文件路径：

- 在 HDFS 中创建 Oozie 文件夹：

```
hdfs  dfs -mkdir /user/oozie
```

- 将文件夹的所有权分配给 Oozie 用户：

```
hdfs  dfs -chown oozie:oozie /user/oozie
```

- 将 Oozie 共享库导入 HDFS：

```
${OOZIE_HOME}/bin/oozie-setup.sh sharelib create -fs
hdfs://<hadoop-namenode-ip>:9000 -locallib
${OOZIE_HOME}/oozie-sharelib-4.3.0.tar.gz
```

13）使用以下命令启动 Oozie 服务器：

```
${OOZIE_HOME}/bin/oozie-run.sh
```

如果 Oozie 正常启动，控制台中应该打印“INFO: Server startup in xx ms”。

然后启动 Hue 控制台并登录以查看 Oozie 登录页面（可通过选择 Workflows | Editors | Workflows 菜单进行访问），如图 11-6 所示。

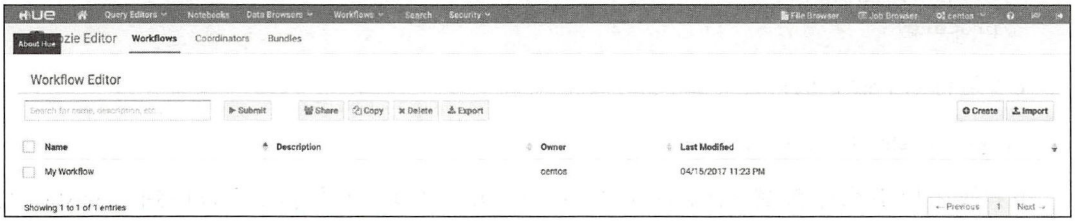


图 11-6 Hue 中的 Oozie 登录页面

3. Oozie 工作流

Oozie 提供了工作流的功能，允许用户定义从一个点到另一个点的处理流程，并支持各种 Hadoop 框架。Oozie 工作流定义了哪些组件可以参与作为独立作业执行的流程。在 Oozie 工作流编辑器中单击 Create（在页面右侧），用户可以直观地创建工作流程。图 11-7 展示了一个工作流的示例。

在这个例子中，我们正在创建一个目录，然后重命名。拖放工具栏中的 HDFS 组件，并输入图 11-7 中的命令。

当用户手动提交执行这个工作流时，界面上会显示执行状态和结果，如图 11-8 所示。

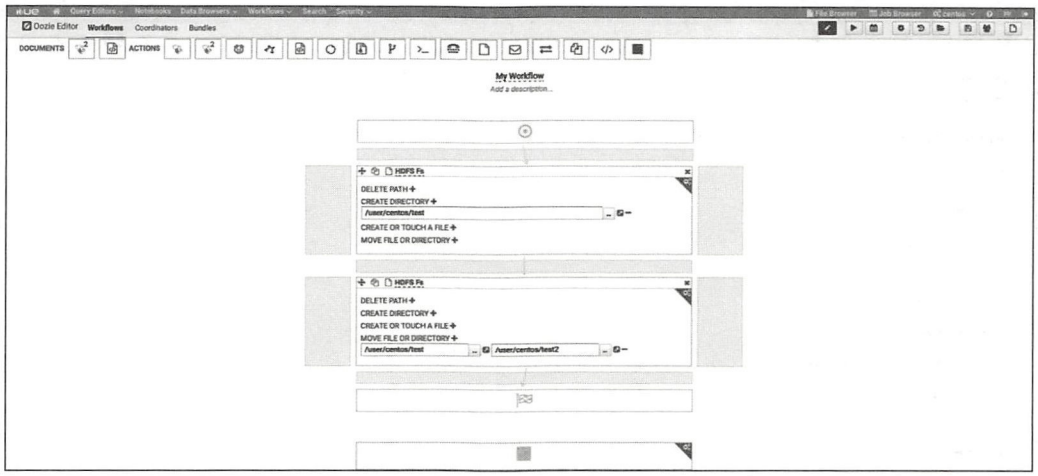


图 11-7　Hue 中的 Oozie 工作流设计器

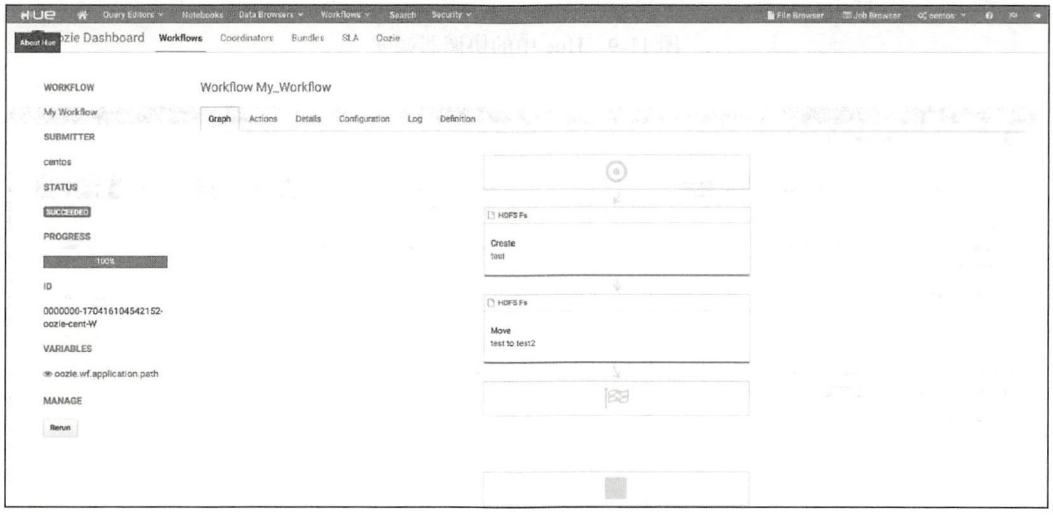


图 11-8　Hue 中显示的 Oozie 执行情况

生产环境下，对于数据持续涌入的系统而言，可能无法手动提交工作流，因此在这种情形下，可能需要调度这些工作流。该功能可以通过 Oozie 协调器来实现，相关内容请参见下面的描述。

4. Oozie 协调器

为了将工作流引入协调器配置中，用户需要创建协调器配置，并在其中加入工作流。为此，用户还可以定义协调器执行工作流的时间表 / 时间间隔，如图 11-9 所示。

Hue UI 还提供了仪表盘来汇总执行结果，提供单个视图显示运行中 / 已完成的作业状态的快照。如果需要，该功能可以帮助用户深入查看各个 Oozie 作业的日志。Hue UI 中的仪表盘如图 11-10 所示。

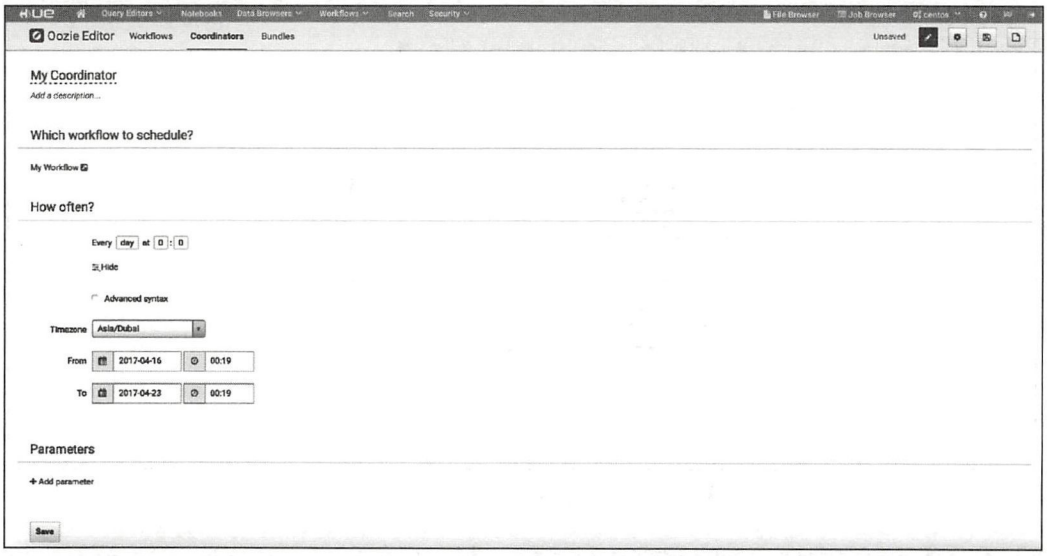


图 11-9 Hue 中的协调器调度

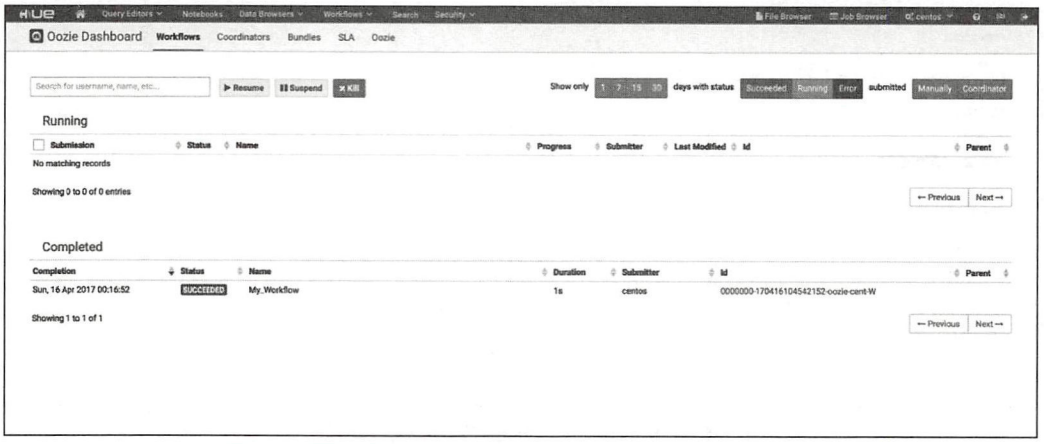


图 11-10 Hue 中的 Oozie 仪表盘

11.7.5 复杂事件处理

第 8 章中介绍 Flink 框架时，提到了 Flink 可以帮助构建多种流式分析中的复杂机制，因为它支持 exactly once 事件处理以及窗口功能。精准数据聚合组件是 CEP 引擎的主要组件之一，只有在事件恰好按 exactly once 处理的情况下才能实现。完整的 CEP 引擎都需要这些模块，在这些特定功能的基础上可以重新构建 CEP 的功能。

但是，要构建完整的 CEP 引擎，需要付出相当大的努力，并且一些用例会要求立即使用完整的 CEP 引擎。对于这样的使用场景，可以将类似 Esper 的嵌入式 CEP 引擎整合到实

仅供非商业用途或交流学习使用

时处理通道中，提供所需的流式数据分析能力。

复杂事件处理包括规则执行、事件关联、基于事件的统计分析和基于模式的多事件分析，以进行近实时的自动建议和决策。

11.8　数据安全

构建企业数据湖，安全性是非常重要的方面之一。数据湖将来自企业的数据集中到一个地方。用户必须征得所有部门的同意才能将数据汇集到数据湖，并且需要保证数据湖中数据的安全，只有通过身份验证和取得授权的人才能访问数据。因此，对数据安全方面需要认真考虑，确保数据是安全的且各业务部门对重要数据的访问规则满意。除了安全设置之外，还应该设置适当的数据治理措施，以提供足够的安全性，但同时要确保对于有访问权限的用户而言，进行深入的分析工作又十分容易。

数据安全，指的是无论是对流式的事务数据还是静态的批量数据的访问和处理，都要经过认证和授权（属性）。

数据湖确实会带来另一种风险，因为它将来自各个孤岛的数据整合到一起，这会产生更多数据安全问题，因为虽然在数据湖中，来自不同系统的数据会整合起来，确实很容易提高分析的丰富性，然而一旦出现安全问题，也会对业务造成不良的影响。

有许多开源项目可以与数据湖集成，以提供各种安全功能，下面各小节中将进行详细介绍。

图 11-11 中以清晰易懂（源自 Hortonworks）的方式显示了 Hadoop 周围的安全环，可以根据企业对数据湖的安全性需求来构建安全环。该图显示了各种安全功能以及对应的技术。

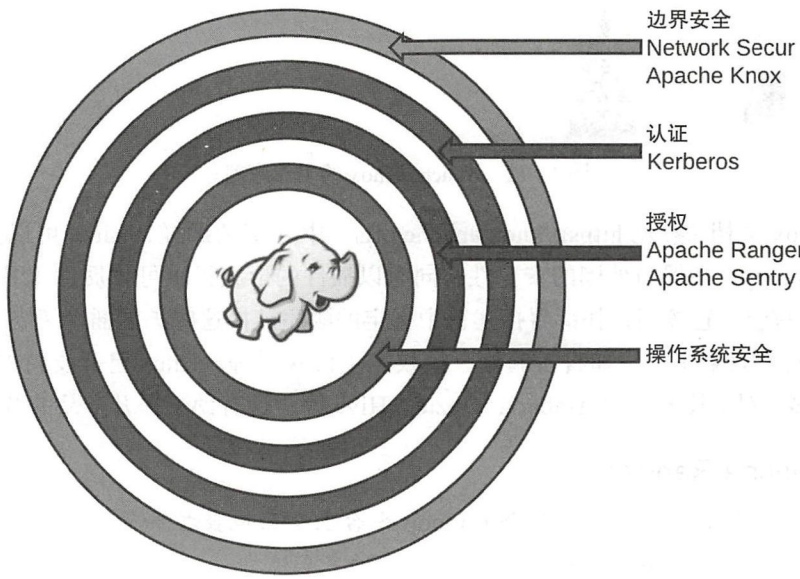


图 11-11　Hadoop 的安全环

仅供非商业用途或交流学习使用

11.8.1 Apache Knox

Apache Knox 网关是一个 REST API 网关，用于与 Apache Hadoop 集群交互。Apache Knox 网关为所有与 Apache Hadoop 集群的 REST 接口进行的交互提供单一访问入口。Apache Knox 网关能够有助于控制、集成、监控以及企业中需要的关键的管理和分析的自动化，这项功能非常有价值。

——https://knox.apache.org/

Hadoop 支持 Kerberos 身份验证，而授权则可以使用其默认的基于 UNIX 文件和目录权限来实现。在 Hadoop 的安全设置中，Apache Knox 可以提供额外的功能。Knox 还可以用于完全没有安全措施的 Hadoop 集群。

图 11-12 显示了 Apache Knox 的工作原理。

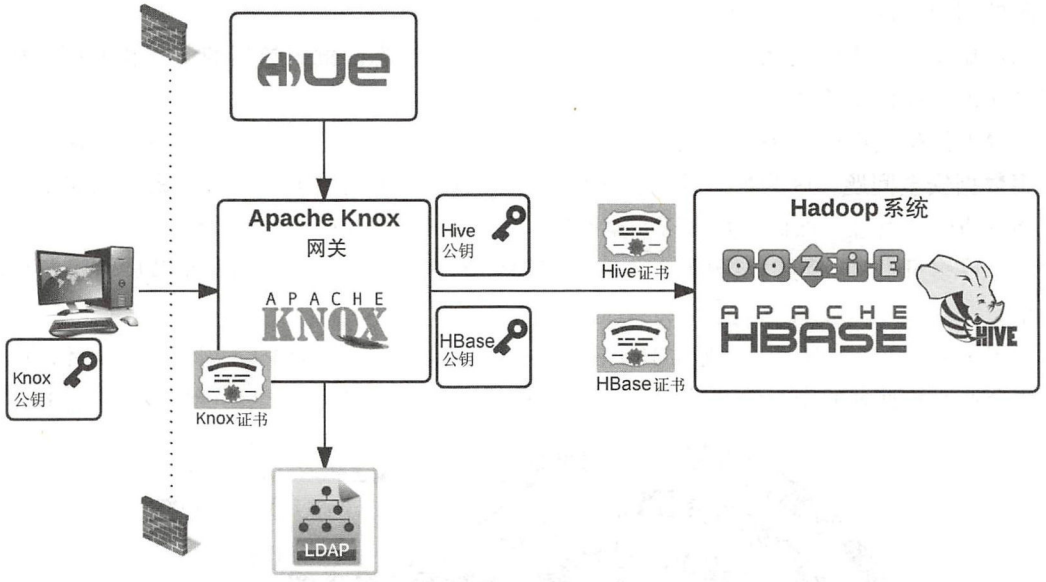


图 11-12 Apache Knox 的工作原理

正如 Knox 文档（参见 https://knox.apache.org/）中介绍的那样，Knox 可以充当反向代理（reverse proxy）来保障外围的安全性，同时以插件的形式提供可扩展性（用于各种安全策略合规性检查）。它按照拓扑部署描述符中指定的顺序链接这些扩展插件（按所需的顺序）来支持各种安全策略实施，如身份验证、授权、主机映射等。Knox 已经支持 Hadoop 生态系统中的许多产品 / 技术，如 Ambari、Oozie、Hive 等，这有助于提升产品的竞争力。

11.8.2 Apache Ranger

Ranger 是一个框架，可以在整个 Hadoop 平台上全面地监控和管理数据安全。Ranger 的愿景是在 Apache Hadoop 生态系统中提供全面的安全性。

——http://ranger.apache.org/

仅供非商业用途或交流学习使用

图 11-13（源自 Hortonworks）显示了 Apache Ranger 的工作原理。Ranger 可以为 Hadoop 生态系统中的各种产品和技术提供授权功能。

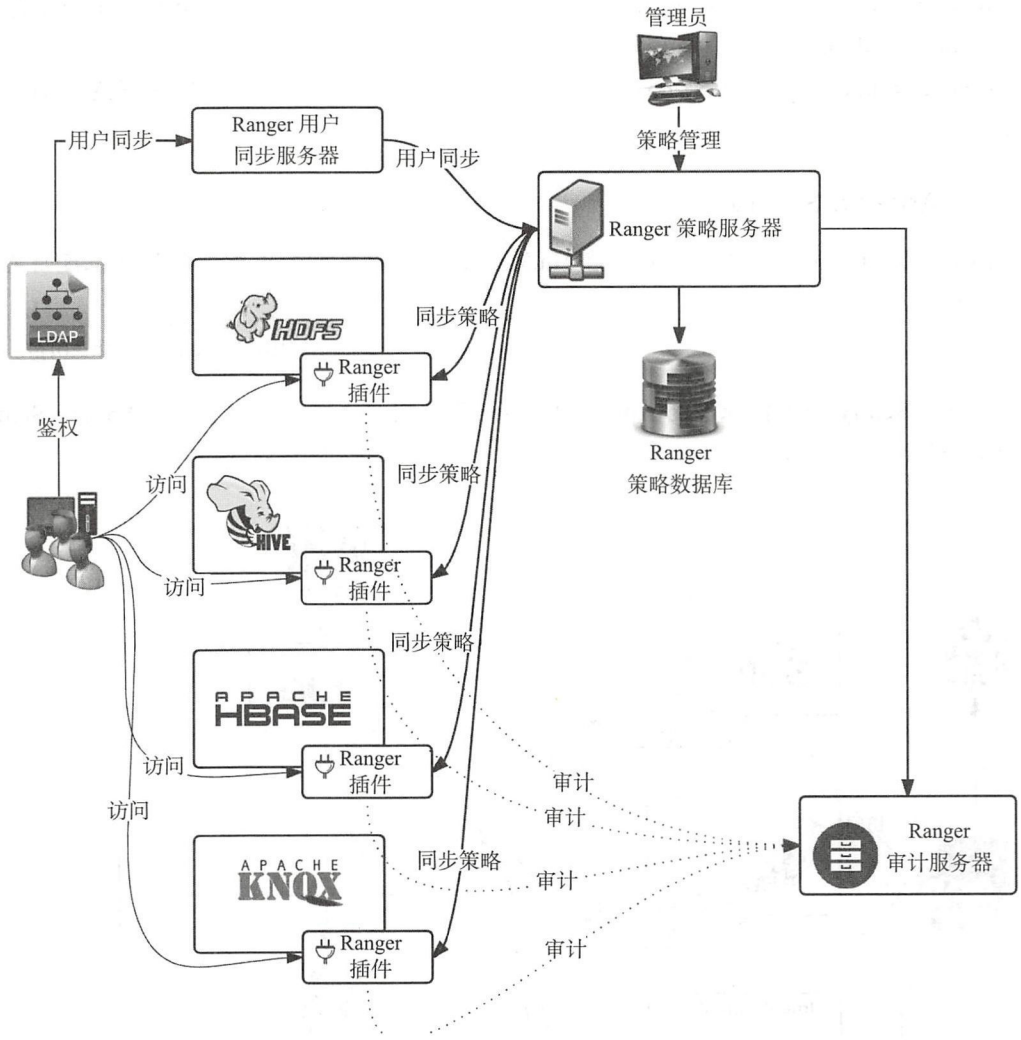


图 11-13　Apache Ranger 的工作原理

Ranger 的授权方法是基于属性的访问控制（Attribute Based Access Control，ABAC）。ABAC 基于 4 个属性，即主体、行动、资源和环境。

如图 11-13 所示，Ranger 插件随产品一起安装，为此需要授权。Ranger 将用户数据与企业目录（存储用户凭据的位置）同步，并利用它们来设置安全策略。这些安全策略由管理员设置和保存。当用户试图访问安装了 Ranger 插件的产品中的数据时，它会检索存储的策略并在用户访问所需的数据之前进行适当的检查。Apache Ranger 在 Hadoop 生态系统中支持 HDFS、Hive、HBase、Storm、Solr、Kafka 和 Knox。

仅供非商业用途或交流学习使用

除了授权（这是核心能力）之外，Ranger 还捕获并存储各种审计活动。当需要对特定的活动进行跟踪时，这些捕获的数据将极具价值。

Apache Ranger 最初由 Hortonworks 开发，至今仍属于 Hortonworks，因此它与 Hortonworks Hadoop 发行版具有良好的兼容性。

Apache Ranger 与 Apache Knox 协同工作，实际上它们以多种方式相互补充来实现安全目标。

11.8.3 Apache Sentry

Apache Sentry 是一种用于对存储在 Hadoop 集群上的数据和元数据进行细粒度角色授权的系统。

——https://sentry.apache.org/

Apache Sentry 的工作原理与 Apache Ranger 非常相似。图 11-14 显示了 Apache Sentry 的工作原理。

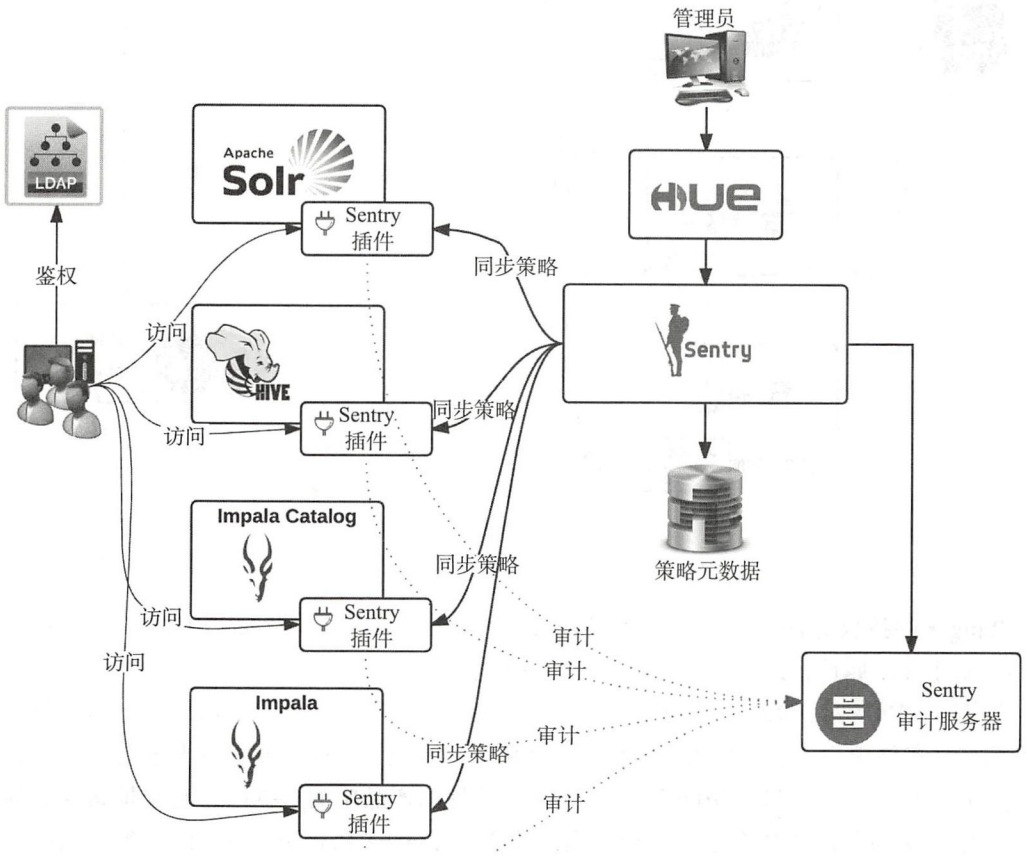


图 11-14 Apache Sentry 的工作原理

仅供非商业用途或交流学习使用

Sentry 插件需要安装在数据处理技术框架（Hive、Impala 等）之上。任何对这些组件中数据的访问首先会被插件拦截，并且只有当它满足 Sentry Server 的元数据中定义的所有安全策略时，才允许访问。Sentry 服务器负责管理所有的授权元数据，并且始终与安装了数据处理技术的插件保持联系。这是 Cloudera 发起的安全项目，这使得它与 Cloudera Hadoop 发行版兼容性更好。Sentry 支持 Hadoop 生态系统中的 HDFS、Hive、Impala 和 Solr。

在建设数据湖的安全性方面，Ranger 和 Sentry 都是不错的选择。如果用户决定使用 Cloudera，自然而然会倾向于选择 Sentry。如果选择 Hortonworks 的 Hadoop 发行版，请选择 Ranger。

11.9　数据加密

数据湖中的数据对组织来说非常关键，必须始终保证它的安全。另外，为了符合组织内部的各种法规和安全策略，认证和授权必须与数据加密一起使用。需要为以下两类数据进行加密：

- 静态的数据
- 传输中的数据

图 11-15 显示了如何使用加密技术来保障静态数据和传输中数据的数据安全。

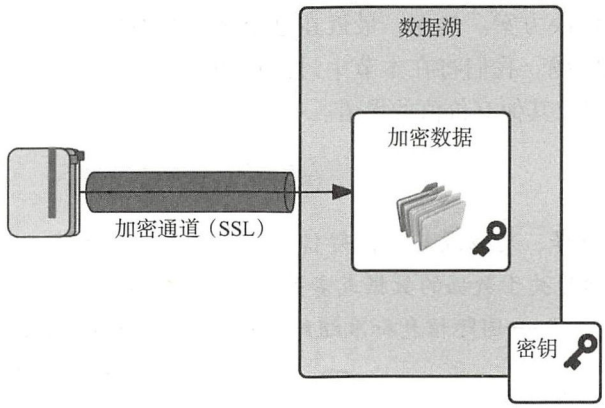


图 11-15　数据加密

在启用身份验证和授权之前，需要传输凭证，因此需要保障传输通道的安全性，这一点非常重要。为此，通道应以数据加密的方式保证数据传输的安全。Hadoop 生态系统中的各种技术使用 RPC、TCP/IP、HTTP 等多种协议相互通信。根据协议的不同，通道保护方法也有所不同。

Hadoop 密钥管理服务

Apache Hadoop 包含了一个内置的密钥管理服务器（KMS），它保障了基于 HTTP 的数

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

据传输，同时提供了客户端、服务端 REST API 来保护通信渠道，如图 11-16 所示。

Hadoop 密钥管理服务器本质上是一个 Jetty 应用程序，它支持容纳多个 Java 密钥存储，还提供了访问和管理密钥元数据的 API。从功能安全角度来看，Hadoop 密钥管理囊括了基于访问控制列表（Access Control List，ACL）的访问管理，并支持多种身份验证和授权协议，如 Kerberos、Active Directory、LDAP 以及基于 SSL 的通道安全。Hadoop KMS 包含端到端的数据加密，覆盖了静态数据和动态数据的加密。一旦数据写入 HDFS，就会使用特定算法对其进行加密，并将其分配给安全区域。

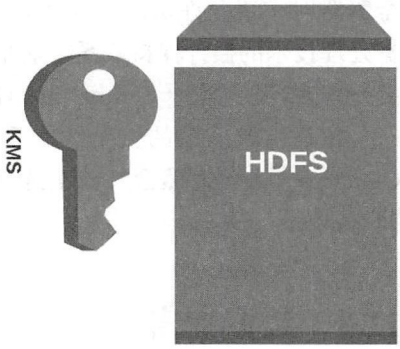


图 11-16 Hadoop 密钥管理服务

11.10 元数据管理和治理

在元数据管理和治理这两个方面，大数据领域中的许多技术都需要创新和发展。已经有一些技术确实提供了这方面的部分功能，但是还相当有限，不足以被认为是适合企业的解决方案。然而，最近在这个领域有不少开发者正在认真地开展工作来解决这两个方面的问题。我们将在本节中讨论这些内容。在进一步研究之前，需要先来详细了解这两个术语以及其他有价值的术语。

11.10.1 元数据

元数据是描述、解释、定位或其他使得信息资源更容易被检索、使用或管理的结构化信息。元数据通常被称为关于数据的数据或者关于信息的信息。

——国际信息标准组织（http://www.niso.org/publications/press/UnderstandingMetadata.pdf）

如维基百科（https://en.wikipedia.org/wiki/Metadata）中所述，元数据分为 3 类：

- 描述型元数据（descriptive metadata）：让资源能够被发现和识别的信息 / 数据属于此类元数据。
- 结构化元数据（structural metadata）：描述复合对象内部的组成和构建情况的信息 / 数据属于此类元数据。
- 管理类元数据（administrative metadata）：有助于管理资源并包含数据属性的信息 / 数据属于此类元数据，如创建对象时的文件类型及其他技术信息。

处理数据时，元数据管理确实是一项非常重要的功能。当数据的量级变大且类型更加

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

多样化时，这种功能变得更加重要。数据湖是基于许多 Hadoop 生态系统技术构建的，具有多样性，因此非常依赖元数据管理。另外，数据湖不会以标准化的方式保存数据，而且各个模型中为了适配应用程序的需求会复制数据副本。对于数据湖来说，这一点非常重要，因为它允许企业确保满足一些重要的非功能性需求，具体如下：

- 数据发现：发现数据及其各类属性。
- 数据质量：数据来源及可靠性。
- 数据可用：这些数据的可用性。

元数据提供了很多便利，但为了利用元数据的价值，需要妥善管理它们，并且必须投入时间来确保其质量（正确性和完整性），这样也提升了可用性。

11.10.2　数据治理

数据治理（data governance）是一种控制手段，可确保运维团队成员或自动化流程输入的数据符合严格的标准，比如业务规则、数据定义和数据模型中的完整性约束等。

数据治理是一组过程，确保重要的数据资产在整个企业中得到正式的管理。数据治理确保数据可信，并且如果由于低质量数据而引发了任何不良事件，会有对应的人员为此负责。

——维基百科

数据治理为数据湖内的数据提供了正确的控制和充分的信任。这使得业务部门可以在数据湖上进行数据分析，从而根据数据湖的建议（分析结果）做出重要的业务决策和流程变更。

数据治理的重要性甚至促使企业在组织中创造了一个新角色——首席数据官（Chief Data Officer，CDO），由这个角色来治理和控制企业中的数据。

11.10.3　数据世系

数据世系（data lineage）被定义为一个数据生命周期，它包括数据的最初形态以及数据是如何随着时间变化的。数据世系描述了数据在经历多次处理时发生了哪些变化，能帮助提供数据分析流水线的可视化功能，简化了当出现错误时溯源的工作。

——维基百科

数据世系是数据流从源头到目的地之间的追踪信息的可视化表示。元数据是实现这种可视化表示的一个关键因素。数据世系还代表了数据流中的各种流程和转换，也能够追踪数据流中的各种依赖关系。

数据世系还标识了企业中特定数据元素的正确来源，企业往往拥有多个可作为数据源的内部系统。它有助于避免数据冗余，并保证数据的质量和完整性。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

11.10.4 如何实现

一些选项可以简单地描述一下。不过这只是笔者的观点，并不是最权威的解释。我们希望能借此传达元数据管理和治理的重要性。

1. Apache Atlas

Atlas 是一个可伸缩、可扩展的核心基础治理服务——能够有效满足企业在 Hadoop 范畴内的监察要求，并且能够与整个企业数据生态系统集成。

——http://atlas.apache.org/

在撰写本书时，Apache Atlas 还是一个由 Hortonworks 开发的 Apache 软件基金会（Apache SoftwareFoundation）的孵化项目。该项目旨在解决 Apache Hadoop 中的数据治理问题，并帮助与组织中的其他企业数据应用程序集成。

Atlas 的架构细节可参考 http://atlas.apache.org/Architecture.htm，图 11-17 所示是它的高层次的概念架构图。

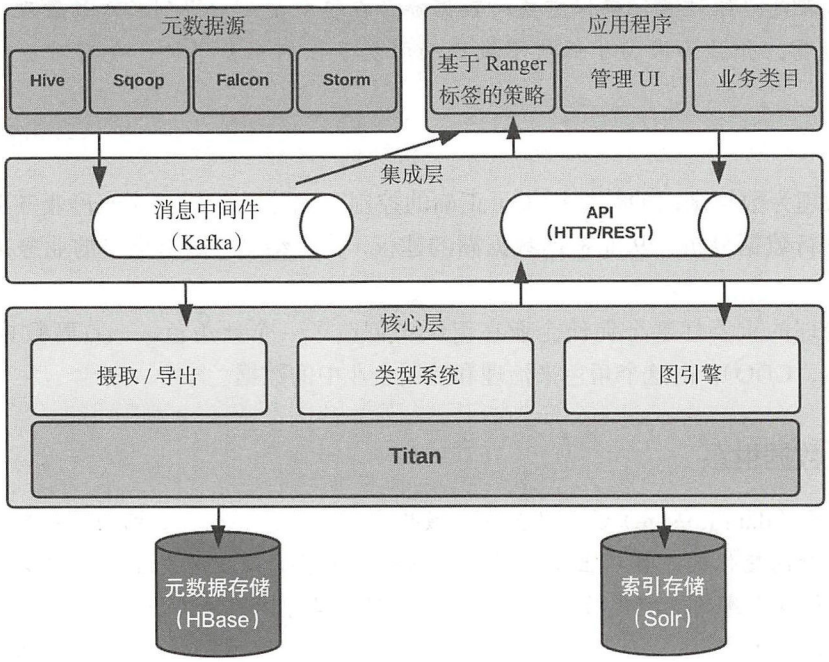


图 11-17 Apache Atlas 高层次概念架构图

在这个领域中，几乎没有什么类似 Informatica 的商业产品，Informatica 提供大数据适配器，可以在信息生命周期中跟踪信息的变化。Cloudera、Hortonworks、MapR 等大数据技术提供商也提供了类似的产品。此类产品能有效帮助企业围绕信息架构和处理流程实施数据治理。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

让我们从下往上快速浏览一下 Atlas 的功能模块。Atlas 核心层由 4 块组成，如下所列：

- 摄取 / 导出（Ingest/Export）：顾名思义，该组件负责摄取和导出元数据。从图 11-17 中可以看到，元数据存储在元数据库（HBase）中。
- 类型系统（Type System）：Atlas 允许定义如何存储元数据的模型。它使用所谓的类型（type）来完成这个操作，该模块正是用来执行这个功能的。
- 图引擎（Graph Engine）：Atlas 的元数据存储在图模型中，该模块专门用于处理图模型。
- Titan：Atlas 使用 Titan（参见 http://titan.thinkaurelius.com/）作为图数据库来存储元数据。

第二层为集成（Integration）层，它的功能是让 Atlas 与外部组件集成。下面列举的是 Atlas 与外界联系的两种方式：

- API：Atlas 的大多数函数向外暴露为 REST API，该模块负责处理此类事务。
- Messaging：Atlas 也可以通过消息中间件与外界联系，经典做法是利用 Kafka 的 topic 与外界交互。

Apache Atlas 天然支持从多种源系统收集元数据，下面列举的是一些 Atlas 支持的比较常见的数据源：

- Hive：http://atlas.apache.org/Bridge-Hive.html
- Sqoop：http://atlas.apache.org/Bridge-Sqoop.html
- Falcon：http://atlas.apache.org/Bridge-Falcon.html
- Storm：http://atlas.apache.org/StormAtlasHook.html

有一些应用程序可以作为 Atlas 对外服务的窗口，列举如下：

- 管理 UI（Admin UI）：这是一个 Web 应用，通过它来管理 Atlas。
- 基于 Ranger 标签的策略（Ranger Tag based Policy）：Ranger 可以集成到 Atlas 中，用于安全策略管理。
- 业务类目（Business Taxonomy）：该组件利用存储在 Atlas 中的元数据与业务对象连接。

2. WhereHows

WhereHows 是一个在 LinkedIn 内部得到广泛应用的元数据管理工具，最近遵循 Apache License 开源，笔者认为可以利用 WhereHows 实现元数据管理和治理。数据湖实现过程中，WhereHows 是一个值得考虑的元数据管理和治理的选项。

WhereHows 是一个 LinkedIn 内部使用的数据发现及数据世系工具。它整合了几乎所有主流的数据处理系统，并从中收集 catalog 及操作型元数据。

—— WhereHows Github project

关于该工具的更多细节，可参考 Github 中的相关介绍（https://github.com/linkedin/WhereHows）。

非卖品，仅供非商业用途或交流学习使用

11.11 数据审计

从数据湖的视角来看，审计（auditing）是一项不可或缺的重要功能。数据从不同的数据源、部门汇集而来，数据资产类型多种多样（机密型、可公开型等），由于这些因素，某些数据需要特殊的安全机制和处理方式。对于数据湖中的一些特殊数据，需要追踪数据的每一次变化以及被谁访问（此时需要考虑法律和契约问题）。

在源系统中，数据被长时间保存，这是进行日常活动（考虑到生产周期）的必要条件。在此之后，数据通常被归类为非生产数据，被归档或离线存储。对于数据湖来说，没有归档数据的概念，因此，需要时时对数据进行访问控制和审计（它经历了各种各样的转换）。当然，并不是所有的数据都需要审计，但是一些数据确实需要，并且必须被审计。

进行数据审计并不简单，但从长远来看，这将带来益处，特别是对于那些被归类为高度机密的数据来说。审计需要捕获旧数据、被变更（新）的数据以及一些元数据，比如对哪些数据做了更改，什么时候更改的，等等。

前面的内容中已经对数据湖中的数据资产进行了详细叙述，可以对数据资产进行分类（如按机密程度高、中、低来划分），然后进行必要的数据审计。一旦启用了审计功能，根据已配置的规则，数据湖应该能够向管理员触发适当的警报，并生成风险提示和法律遵从性的报告。

拥有了前面介绍的所有功能，便能满足审计需求。回顾一下，它们分别是：

- 适当的访问控制。
- 数据变化追踪。
- 基于已有配置触发风险报告的能力。

对 Apache Atlas 及 Apache Ranger 进行配置、使用，将能给我们带来所期望的数据审计能力。

Atlas 执行必要的数据审计，Ranger 负责数据湖中的数据授权方面。

11.12 数据可追溯性

可追溯性指的是借助记录标识来确定数据项的历史、位置、应用的能力。

——维基百科

数据可追溯性指的是可明确数据在从一个位置（源）移动到另一个位置（目的地）的路径，以及在到达目的地之前进行的各种过程和转换。我们已经了解了什么是数据世系，那么数据世系和数据可追溯性之间的区别是什么呢？

数据世系通常与元数据管理和治理相关联，与数据可追溯性的含义有很大的不同。

数据世系更偏重技术，显示了数据从起点移动到终点的每一个重要步骤。对一个技术团队来说，这是一个非常重要的能力 / 资源，但是它对企业的非技术业务及其他用户没有太

非卖品，仅供非商业用途或交流学习使用

大的意义。

数据可追溯性引入了一个非技术层级，该层为企业中各类用户引入了很多与技术无关的操作模式。

并不存在一个工具可以全自动地完成所有这些工作，需要跨部门、多人员的协同合作，对数据可追溯性进行维护和管理，因此，企业中每个人进行决策时，需要意识到对数据的影响，以及对其他部门的依赖。

11.13　了解更多服务层细节

数据湖中负责与外界交互的是服务层。该层为不同的用户提供数据服务以满足其需求。下面将简要讨论一些该层需要重点关注的问题。当然，该层也会采用一些技术来帮助提供数据给最终用户。其中大多数是与数据持久化存储相关的，包括关系数据库、NoSQL 数据库、文档存储、键 / 值存储、列式数据库等。

11.13.1　服务层构建原则

在本书的第一部分已经对服务层做了一些介绍。这里只是回顾一下，因为技术框架的选择是由构建原则驱动的。

- 快速访问 / 高性能（fast access/high performance）：能高频次地为用户提供服务。
- 读取、更新低延迟（low latence reads and update）：数据读取、更新操作延迟尽可能低，这样能提高服务响应速度。
- 支持随机读写（capability of random read）：具有支持随机读取的索引能力，可以在海量数据集中查找某个子集。
- 高度可伸缩（high scalability）：服务层是数据湖对外服务的窗口，因此它必须高度可伸缩，用来服务多个用例和用户。
- 容错（fault tolerant）：在生产运营期间，企业对数据湖的依赖程度会日渐加深，因此，它必须具有容错能力才能继续为用户及其用例服务。
- 多模型服务能力（capable of serving multiple model）：数据模型非规范化，通过预先计算与存储，为终端用户提供灵活的数据服务。

11.13.2　服务类型

在服务层，有多种方式可对外暴露数据，其中一种方式就是在 HTTP（HTTPS）上利用 REST（Representational State Transfer）端点。数据服务实现者也可以利用 SOAP（Simple Object Access Protocol）来对外暴露服务。哪种选择更好？ REST 还是 SOAP？在这里并没有标准答案。我们使用 REST 而不是 SOAP，主要是基于 REST 的下列优点：

- 相比 SOAP，REST 更容易掌握。

非卖品，仅供非商业用途或交流学习使用

- REST 可以以多种数据格式向外暴露服务，JSON 是其中之一，它被认为是轻量级的，易于在网络上传输，解析更快、更简单。
- 能很好地与网络协议（如 HTTP）交互。
- 使用基于 HTTP 的 RESTful 服务，能享受 HTTP 的所有优点：
  - 能被多种协议理解。
  - 可以被缓存（很多协议默认支持缓存）。
  - 借助 SSL/TLS 保障网络安全。
  - 可利用 PUT、PATCH、POST、GET、DELETE 等默认 HTTP 方法来处理资源。
  - 可使用多种加密技术，这方面已经非常成熟了。

可以从服务层对外暴露多种服务。可以将服务分为两类：数据服务和业务服务。

1. GraphQL

既然已经讨论了为什么选择 REST 而不是 SOAP，那么也非常有必要对 GraphQL 展开简短的讨论。

数据湖的主要使命之一是满足各种用户的需求。每个使用者对数据属性和数据格式有不同的要求。通常，根据消费者的需求，需要创建很多端点。在生成周期内，端点数量可能会快速增长并很快为维护带来问题。

GraphQL 部分解决了这个问题，服务实现者不需要根据面向消费者的模型的属性需求来编写大量不同的服务。

GraphQL 是一种 API 查询语言，以及利用现有数据填充查询的运行时环境。GraphQL 为 API 中的数据提供了完备的、易于理解的描述，它非常强大，能帮助客户端精准请求所需数据，轻而易举地解析随时间演化的生产环境中的 API，也能为开发工具赋能。

—— http://graphql.org/

GraphQL 确实可以被认为是在现有的 REST 端点之上构建的另一层，用来满足消费者需求的模型的各种要求（特别是从模型中过滤属性）。

2. 数据湖与 REST API

数据湖中的各种数据模型可以以 JSON 格式通过基于 HTTP（S）的 RESTful 端点对外公开。这是我们为最终用户提供服务的主要方式。

记录 Web 服务是一项相当乏味的工作。如果这些公开的 Web 服务没有被记录，那么数据湖很快就会变成一个“黑盒子”，如果没有人知道里面到底有什么，就不会被人们充分利用了。

Swagger 是一个非常强大的开源框架，能让使用者以非常轻松的方式来记录 REST 端点。

Swagger 是一个非常强大的开源框架，它背后有很大的工具生态系统提供支持，能帮助使用者设计、构建、记录、消费 RESTful API。

—— http://swagger.io/

非卖品，仅供非商业用途或交流学习使用

使用 Swagger 可以很容易地记录每一个 API。如果读者所在企业已经存在一个 API 网关，那么可以非常轻松地在 API 网关中对外公开 API，同时也能使用 Swagger 中的记录，因为大多数网关都支持 Swagger。

3. 商业服务

数据湖从来不会被当作一个事务系统的存储工具（持久化机制）。然而，OLTP 系统可以利用数据湖对外暴露的各种业务服务来满足用户需求。业务服务通常通过消费多个数据服务来提供某种业务功能，而数据服务执行数据层面的操作，确保数据平台中的数据以最自然的方式对外暴露，而对业务逻辑或业务处理没有任何影响。业务逻辑和业务处理应该发生在业务服务层，使我们能够实现松散耦合的服务生态系统，并让数据服务保持最纯粹的形式。

下面是一个关于业务服务的例子：

假设有一个用于销售产品的 OLTP 应用程序。已经从数据湖中抽取出了大量的数据，并建立了分析逻辑来为客户寻找值得推荐的产品。该产品推荐功能可以作为业务端点（基于 HTTP 的 REST）对外暴露，当客户在线时，OLTP 应用可以通过访问该端点来向客户展示推荐的商品。数据湖中的推荐分析模块可以同时利用真实客户的旧的行为数据（这些数据是历史数据，离线存储在 Lambda 批处理层）及新产生的数据（当前的事务数据，保存在 Lambda 快速处理层，由 OLTP 实时产生）。

11.13.3　服务层组件

在数据湖中，实时的事务数据与历史的事务数据存储方式是一致的。数据湖中的大多数数据分析同时需要用到这两种数据以获取有意义的业务洞察力。

在数据湖中，用户可以要求针对当前 OLTP 数据（实时数据服务）以及历史 OLTP 数据（批处理数据服务）提供定制的服务。在 Lambda 架构体系中，这些是通过快速处理层和批处理层来实现的。

1. 数据服务

正如前面所讨论的，数据服务提供了一种基于轻量级协议的、将满足数据契约的数据传输给消费者应用的机制。该组件存在于数据湖的服务层中，传输机制为由消费者应用触发的拉取机制。虽然数据可以近实时地反映到底层数据存储中，但这也取决于消费应用何时通过服务拉取数据。这些轻量级的服务可以满足来自数据湖服务层的实时需求，服务层角色通常由快速处理层来扮演。

Elasticsearch 是一种可用于快速查找和近实时查询的数据存储，并且可以用来构成服务层的一部分。我们这里所说的实时性是指文档一旦被摄取就可以进行检索。Elasticsearch 支持实时的基于关键字的文档查找，也支持数据服务及其数据契约所需的近实时搜索。当数据存储满足数据服务的数据需求时，理想情况下，该存储层应该被放置在数据服务的后面。

非卖品，仅供非商业用途或交流学习使用

Elasticsearch 满足最终一致性，不支持 ACID 事务，但支持轻量级的满足最低限度事务一致性的乐观锁。该选项适合需要部分匹配和搜索的场景，并且需要完整的数据文档。

其他选项可能包括 NoSQL 存储，如 HBase 和 Cassandra，它们被证明适用于基于 Key 的快速查找，也易于被 Java 应用程序调用。HBase 和 Cassandra 都提供基于列族的存储，有助于实现实时检索场景中的基于 Key 的快速查找。HBase 的主要优点是它工作于 HDFS 之上，而不像 Elasticsearch 需要直接与存储打交道。

虽然人们大多数时候将 Apache Hive 看作一个数据湖存储和查询组件，但是如果通过 Hive 视图暴露的数据是模型数据，并被其他应用程序消耗，那么它也可以扮演服务层中的角色。数据推送和数据拉取这两种范式都可能存在。由于 Hive 支持通过 JDBC 驱动程序进行访问，其他应用程序可以通过 JDBC 通道来拉取处理过的信息。此外，由于它是 Hadoop 存储层的一部分，因此可以将 ETL 模块嵌入适当位置，以便适时将包含建模数据的数据从 Hive 视图中推送出来。

Apache Impala 是通用 SQL 查询引擎（也称为 Hadoop 的交互式 SQL），对我们的数据湖实现来说是一个非常合适的补充。内置支持多种 Hadoop 文件格式（如 Avro、Parquet 等），运行速度非常快——这些是将 Impala 作为技术选项的主要原因。

Impala 由 Cloudera（基于谷歌 2010 出版的 Dremel Paper）开发的，然后开源到 ASF，现在正在与 2.7.0 版本一起孵化。它在 Hadoop 生态系统中被广泛支持和集成。更多细节请参考 http://impal@.apach.org/ 和 https://github.com/apache/incubator-impala。Impala 具有高性能（在内存查询和直接访问数据）、高灵活性和水平可伸缩性，可根据需要添加更多的节点。出于这些好处，它与大多数商业 Hadoop 发行版一起发布或打包。

强烈建议将此技术视为对数据湖实现的绝佳补充，以便对大型 Hadoop 数据集进行快速分析查询。Impala 利用相同的 Hive 元数据库，并可以通过 Impala 表对底层 Hadoop 存储进行并行查询。也可以使用 JDBC 驱动程序访问 Impala 表，这使得它们可以用于响应消费者应用程序的数据访问。所有这些都与优化后的 I/O 组合使用，帮助查询更快地运行，避免投资昂贵的 ETL 工具。相比于 Hive，Impala 提供了相对更实时的查询，同时利用与 Hive 相同的共享资源来运行较短的作业。

根据处理数据的作业的运行时间长短，可以选择 Hive 或 Impala 视图，但是两者都具有非常相似的数据服务机制。

2. RDBMS

将 RDBMS 作为潜在的服务层组件似乎是出乎意料的，但是，可以服务于消费者应用的任何组件都可以作为服务层组件。很多时候，消费者可能希望使用关系模型来访问数据，而关系模型只能在关系数据库中持久存储。数据湖也可能需要将处理后和建模后的数据存储到关系数据库中，并将数据视图暴露给消费者应用程序。这里需要考虑的是，传统的关系数据库并不是为大数据场景构建的，因此服务层组件中的数据应该尽可能简洁，并对消费者应用程序在访问数据时的操作进行限制。

11.13.4　数据导出

使用 REST 端点（Web 服务）对外暴露数据，对于许多用例来说是非常有用的而且是必备的功能，但是有一些用例需要以更为有序的方式从数据湖中获取数据，即便是这样，也需要相当大的数据量。在这种情况下，用户甚至可以通过 Apache Sqoop 来将数据从数据湖中对外暴露或传输到其他消费应用程序的数据存储区。其他机制可能包括调度 MapReduce 作业执行 ETL 操作，并将处理结果迁移到 Hadoop 外部，必要时会触发一些操作，如使用脚本或 ETL 工具将数据推到 ftp 位置，这些脚本或 ETL 工具支持 HDFS 集成，如 Talend 和 Pentaho 的数据集成。

这样的导出最好是批量进行存储，因为批量存储是为大型批处理设计的。在数据存储中运行这样的作业，意味着在近实时场景中会影响这些系统的响应性和违背它们的设计初衷。

11.13.5　混合数据访问

混合（polyglot）在计算机领域是一个非常常见的术语，它与 multiple 等同。

在服务层中，会有多种数据存储（持久性机制）用于存储同一份数据对应的不同的模型，这取决于具体的用例。我们并不能确定工业界是否存在混合数据访问（polyglot data access）这样的术语。通过混合数据访问，即在服务层中存在多种数据存储，服务层使用它来为终端用户提供形式多样的数据服务。

在我们的数据湖实现中，使用了 Elasticsearch、Hive 和 HBase 作为数据访问的存储组件，在实际操作中使用了混合数据访问。

混合数据访问的概念可以进一步扩展，用来表述一个对象持有多个待持久化存储的信息子集，它们分别被存储到不同的存储系统，并从这些存储系统中被检索。现在也有一些框架在大数据技术中支持这种机制。

混合数据访问的一个非常突出的案例是索引和数据分别存储，这样，索引技术主要用于索引，而数据技术则主要致力于提高存储效率。在大数据背景中，混合持久化框架有许多选项，JPA 接口结合 Hibernate 使开发人员能够使用标准的持久化配置和注解来定义对象。

11.13.6　范例——服务层

现在来看一个服务层与数据设备的例子，正如本章已论过的那样。人们期望数据服务将服务层数据库中的数据传递给请求者，因此这些数据库应支持快速随机访问，Elasticsearch 应该是其中的一种。

上一章中，我们摄取了多种数据（客户信息、地址、联系方式）到 Elasticsearch 中，可基于这些来构建数据服务。例如，可以基于 Spring Boot 构建 JAX-RS 2.0 REST 数据服务，这里可以利用 Swagger UI 来提供服务定义。

源代码仓库中名为 chapter11 的工程中包含了这个例子，可以复用本书前面例子中一直使用的那个数据模型。本范例有一些值得注意的地方：JAX-RS 2 注解，读者可以对它做快速了解，另外就是使用了 Swagger 注解的主服务实现。

这样的数据服务实现极其灵活自由，可以根据需要对数据、单一客户视图进行组合。在这里可以看到，客户服务对客户地址、客户联系方式等隐私数据进行了组合，然后发送响应给请求者。

下面的代码揭示了如何使用 Spring Boot 以及 SpringMVC-Jersey 桥接来实现本范例中的服务，这里可以利用标准的 JAX-RS 2.0 注解。如果读者对基于 Spring 框架的应用比较陌生，建议只运行本范例来了解服务的实现原理。读者应关注更重要的部分：作为 REST 端点的数据服务以及如何利用 Swagger UI 来描述此类服务。代码如下所示，可参考 chapter11/src/main/java/com/laketravels/ch11/service/endpoint/CustomerEndpoint.java。

```
@Controller
@EnableAutoConfiguration
@Api(value = "customer", description = "This is the customer resource that
provides"
        + " customer information ")
@Path("customer")
public class CustomerEndpoint {

private ObjectMapper MAPPER  = new ObjectMapper();

public CustomerEndpoint() {
MAPPER.setSerializationInclusion(JsonInclude.Include.NON_NULL);
MAPPER.configure(MapperFeature.ACCEPT_CASE_INSENSITIVE_PROPERTIES, true);
MAPPER.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false);
MAPPER.setDateFormat(new SimpleDateFormat("yyyy-MM-dd"));
    }

@Autowired
    private ESUtil esUtil;


private Logger logger = LoggerFactory.getLogger(CustomerEndpoint.class);


@Path("{customerId}")
@GET
    @ApiOperation(value = "profile", consumes = MediaType.TEXT_PLAIN,
produces = MediaType.APPLICATION_JSON,
            response = Customer.class,
            notes = "This endpoint is used to get customer profile")
public Customer getCustomer(@PathParam("customerId") String customerId)
throws IOException, ParseException {
return getCustomerProfile(customerId);
    }

@Path("{customerId}/address")
@GET
    @ApiOperation(value = "address", consumes = MediaType.TEXT_PLAIN,
```

```
produces = MediaType.APPLICATION_JSON,
            response = Address.class,
            notes = "This endpoint is used to get customer address")
public Address getAddress(@ApiParam(
            name = "customerId",
            required = true,
            value = "Contains customer Id"
    ) @PathParam("customerId") String customerId) throws IOException {
return getCustomerAddress(customerId);
    }

@Path("{customerId}/contact")
@GET
    @ApiModelProperty(required = true, dataType =
"org.joda.time.LocalDate")
@ApiOperation(value = "contact", consumes = MediaType.TEXT_PLAIN, produces
= MediaType.APPLICATION_JSON,
            response = Contact.class,
            notes = "This endpoint is used to get customer contact")
public Contact getContact(@ApiParam(
            name = "customerId",
            required = true,
            value = "Contains customer Id"
    ) @PathParam("customerId") String customerId) throws IOException {
return getCustomerContact(customerId);
    }

private Customer getCustomerProfile(String customerId) throws IOException {
        SearchRequestBuilder reqBuilder =
esUtil.getConnection().prepareSearch();
        reqBuilder.setIndices("customer").setTypes("customer");
        TermQueryBuilder termQueryBuilder = QueryBuilders.termQuery("id",
customerId);
        reqBuilder.setQuery(termQueryBuilder);
        SearchHits hits = reqBuilder.get().getHits();
long numHits = hits.totalHits();
        Customer customer = null;
if (numHits > 0) {
            customer = MAPPER.readValue(hits.getAt(0).getSourceAsString(),
                    Customer.class);
            customer.setAddress(getCustomerAddress(customerId));
            customer.setContact(getCustomerContact(customerId));
        }
return customer;
    }

private Address getCustomerAddress(String customerId) throws IOException {
        SearchRequestBuilder reqBuilder =
esUtil.getConnection().prepareSearch();
        reqBuilder.setIndices("address").setTypes("address");
        TermQueryBuilder termQueryBuilder = QueryBuilders.termQuery("id",
customerId);
        reqBuilder.setQuery(termQueryBuilder);
        SearchHits hits = reqBuilder.get().getHits();
long numHits = hits.totalHits();
        Address address = null;
if (numHits > 0) {
```

```
            address = MAPPER.readValue(hits.getAt(0).getSourceAsString(),
                    Address.class);
        }
return address;
    }

private Contact getCustomerContact(String customerId) throws IOException {
        SearchRequestBuilder reqBuilder =
esUtil.getConnection().prepareSearch();
        reqBuilder.setIndices("contacts").setTypes("contact");
        TermQueryBuilder termQueryBuilder = QueryBuilders.termQuery("id",
customerId);
        reqBuilder.setQuery(termQueryBuilder);
        SearchHits hits = reqBuilder.get().getHits();
long numHits = hits.totalHits();
        Contact contact = null;
if (numHits > 0) {
            contact = MAPPER.readValue(hits.getAt(0).getSourceAsString(),
                    Contact.class);
        }
return contact;
    }
}
```

如果来源仓库中一开始没有编译，则请运行下面的命令来编译项目。应确保编译本工程之前 chapter08/data-model 已编译。

```
mvn install
```

一旦编译完毕，请根据自己的运行环境配置 chapter11/config/dev.yml。在 chapter11 工程目录下运行下面的命令：

```
java -jar target/chapter11-1.0-SNAPSHOT.jar -DconfigFile=config/dev.yml -
Dspring.logging.config=config/logback.xml
```

一旦 Spring boot server 初始化完毕，请登录 http://localhost:8080/swagger/index.html 打开 Swagger UI。图 11-18 展示了 Swagger UI 的登录页。

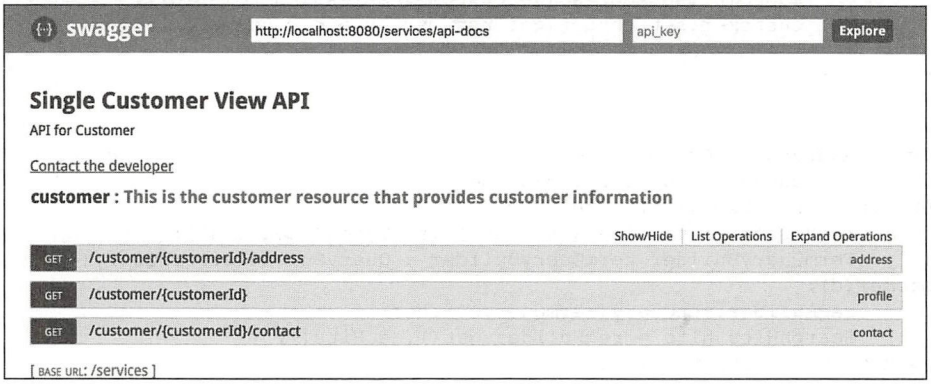


图 11-18 Swagger UI 登录页

在 Swaggger UI 中，尝试调用 customer 端点，紧接着会显示 customer id，读者将看到一个与图 11-19 所示类似的界面——Swagger UI，通过它可以看到数据湖中的数据。对于非技术人员来说，这是一个非常重要的组件，应该在数据湖实现中考虑使用它。

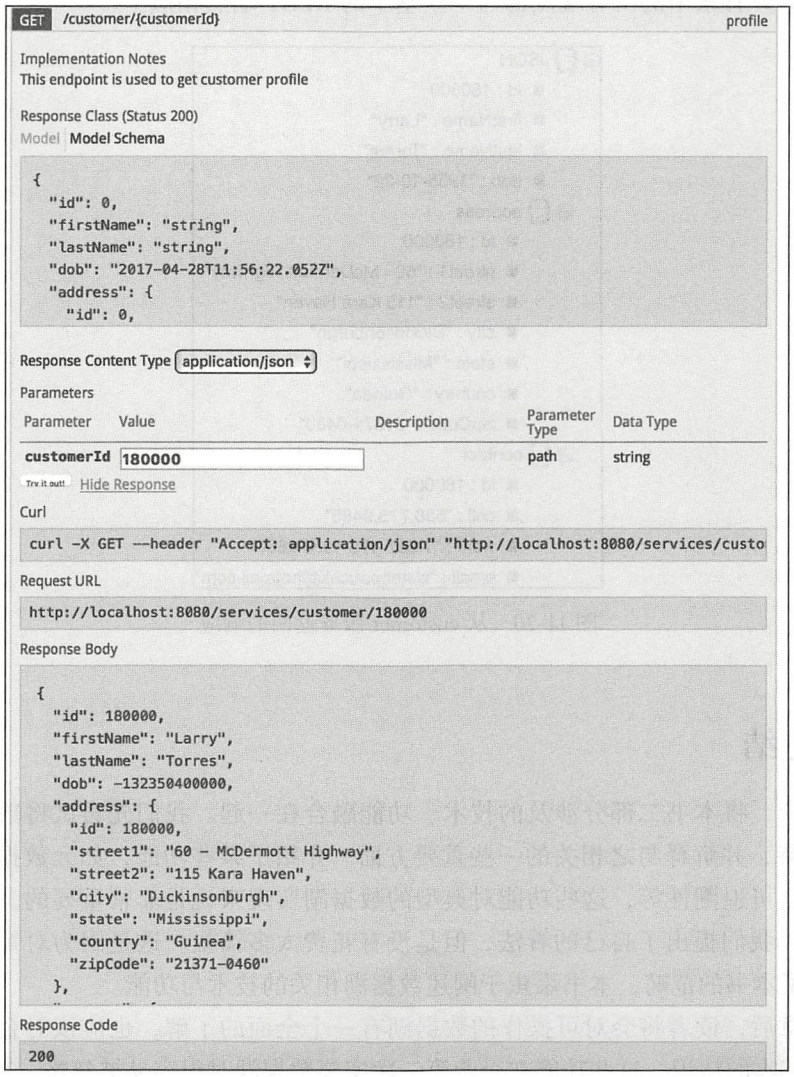


图 11-19　Swagger UI 中的 customer 服务返回的响应

服务的完整响应如图 11-20 所示。

前面的范例中，覆盖了以下内容：

- 构建了 REST 服务端点，用于获取 customer 数据及相关实体。
- 所有的服务端点使用 HTTP GET 方法，这意味着它们是只读型服务。当然，服务端点也可以使用 POST 方法，此时可以在请求体中携带数据回传给服务端，当需要输

仅供非商业用途或交流学习使用

入大量参数时，POST 方法非常有用。

- Swagger UI 通过端点类中的注解和服务接口来推导模型和模型模式，对应的类名是 com.laketravels.ch11.service.endpoint.CustomerEndpoint。
- 特定客户背景中的所有端点都与一些基本的 REST 端点相似。

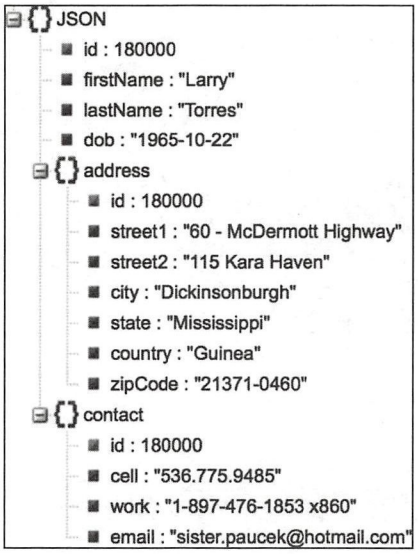


图 11-20 从 customer 服务返回的响应

11.14 总结

在本章中，将本书二部分涉及的技术、功能融合在一起。我们先尝试将数据湖作为一个整体来看待，并解释与之相关的一些重要方面。介绍了某些功能，如元数据管理、数据治理、审计、可追溯性等，这些功能对典型的数据湖实现来说是非常重要的。对很多功能及技术选项，我们提出了自己的看法，但是没有花费太多篇幅。这是因为对每类技术的深入研究超出了本书的范畴，本书聚焦于阐述数据湖相关的技术与功能。

阅读本章后，读者将会对可操作的数据湖有一个全面的了解，也应该对企业数据湖的其他功能有基本的认识，这些功能在企业第一次实施数据湖时很容易被忽略。

真正的数据湖需要这些额外的能力，考虑到本书内容的完备性及篇幅限制，必须提及这些话题，但是又要避免陷入细节的泥潭。在本章中并没有涉及很多代码。一些技术的选择只是我们的观点。读者在实践过程中需要根据实际需要进行慎重的选择。话虽如此，还是鼓励读者在数据湖中实现这些能力，而不是忽略这些能力。

本章是一个不错尝试，因为简单地涵盖了许多不同的方面，相信读者阅读之后有点疲惫了，休息一下，接着学习本书的最后一章。

仅供非商业用途或交流学习使用

第 12 章 Chapter 12

数据湖用例建议

对笔者来说，撰写本书自然会花费相当的精力，对读者来说，通读本书并领会笔者对每个技术点的解释同样也是一大挑战。为了掌握本书内容，读者还需要进行编程实践以便真正掌握每一个知识点。在这里，笔者也应对坚持阅读到此处的读者表示感激，通读本书以及亲自动手实现一个数据湖对读者来说是一个巨大的挑战，因为涉及太多技术框架。向坚持到最后的读者致敬！希望每一页都能给读者带来收获和喜悦，就像笔者撰写此书时的心情一样。

本章旨在介绍一些著名的企业数据湖用例。许多公司已经具备某种形式的数据湖，而有些公司正在进行数据湖建设尝试。

这些用例可以看作我们在组织中证明数据湖的能力的一种方式，然后使用这些成功的故事来继续构建新的用例。

这些建议在本质上来讲并不是面面俱到的，但是我们确实觉得如果读者已经阅读了本书的第一部分和第二部分，就能够很好地执行这些用例，而不会有太多的麻烦。数据湖落地时有变化的地方通常是源数据，以及企业想在每个用例中执行的分析。从业务的角度来看，业务领域的不同可能会导致不同的处理需求。

12.1 在企业中推行网络安全实践

数据量每天都在增长，人们变得越来越精通技术，蠕虫和病毒出现的频率越来越高，破坏力越来越强。在这样的数字化环境中，企业需要对其所持有的数据和业务相关的核心应用程序更加警惕。

仅供非商业用途或交流学习使用

之前企业中孤立的安全团队正向成熟的运营中心转型。这种团队被授权对企业内数据和所有应用程序进行安全审查。

正是有了这样一个不断增长的数据和应用程序使用场景，拥有一个监视各种安全问题的平台显得尤为必要。正是这一要求，使得构建数据湖时增加安全方面的需求变得不可避免。

数据从各种各样的来源进入数据湖，然后由分析人员进行各种分析，用于创建有意义的可视化输出，并主动地根据某些变化采取行动。

拥有一个中央安全监控中心有助于更早地识别漏洞，并可掌握采取安全措施的主动权。可收集数据进行网络分析，对安全合规性及各种操作活动（与安全更相关）进行评估，并根据企业定义的安全策略进行处理。

安全分析师可以查看各种数据，以一种更有意义的方式进行可视化，然后采取必要的行动，以相比事后处置更主动的方式来处理潜在的威胁。将一个安全数据湖（security lake）与其他软件集成在一起，可以根据在数据湖中配置的规则触发各种报警机制。可以在新抵达数据湖的数据之上应用各种复杂的事件处理技术和机器学习技术，以进行额外的分析，这样可以再次为安全分析师主动观察和行动创造机会。

Apache Metron 是 ASF（Apache 软件基金会）的孵化项目，由 Hortonworks 赞助，它利用了之前讨论过的数据湖技术，尤其是在安全领域。建立安全数据湖的方法与普通数据湖非常相似，但是需要从各种数据源收集安全相关数据，这些不同来源的数据可以使用各种工具轻松进行处理，然后 Apache Metron 对数据进行安全相关的威胁检测。在内部，Apache Metron 使用了很多本书第二部分中详细介绍过的技术，相信读者已经非常熟悉了。关于 Apache Metron 的更多细节，建议读者参考 http://metron.apache.org 和 https://cwiki.apache.org/confluence/display/METRON/Metron+Wiki。同样地，在 Hortonworks 官网中也可以了解很多技术细节，网址为 https://hortonworks.com/apache/metron/。

企业网络安全能力构建涉及的领域广泛，如入侵/异常检测、威胁级别的识别、对威胁警报的响应、执行关键保护活动以及威胁消除后的恢复或建立隔离机制。图 12-1 展示了网络安全的生命周期。

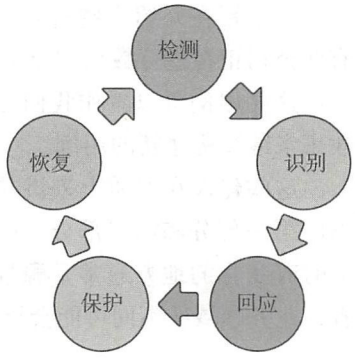


图 12-1 网络安全生命周期

12.2 深入了解企业的客户

在数字转型和数字颠覆的时代，了解每个客户及其偏好是非常关键的。很显然，利用客户偏好信息，构建定制型产品为客户提供个性化服务是企业必备的能力。

通常在企业中，客户数据存在于零散的业务软件中，它们之间不一定需要交换数据。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

我们所说的单一客户视图（Single Customer View，SCV）是指将与客户相关的数据从多个数据源汇总到一起，并基于汇总数据创建单个视图的方法。事实上这还不够，因为客户往往期望更多，这就要求使用更个性化的数据来丰富用户数据，以便分析各种细节，并为客户提供具有针对性和个性化的产品和服务。这会让客户觉得产品与服务与自己息息相关，难以抗拒，因为是针对单个客户的，而不是为一组客户提供的通用型产品和服务。

一个能深入了解客户及其需求的数据湖，对于任何业务来说都是一笔巨大的财富，而应用程序可以寻求数据湖服务层的帮助，从而获得针对客户的洞察力，进而为客户提供个性化服务。

拥有一个能够了解客户及其需求的数据湖，可以让客户时时刻刻得到全方位的体验，无论他们是在台式计算机上使用 Web 应用程序，或是在移动设备上使用移动 Web 应用程序，还是在智能手机上使用移动应用程序。集中的数据湖是用户体验一致性的有力支撑，同时也避免了重复的广告活动及广告目标。这里的意思是，如果客户在使用 Web 应用程序时表示他们不想看到特定类型的广告，那么他们也期望商家在其他客户交互渠道中也应对此有所反馈。客户交互渠道与客户关系紧密，而数据湖中存储了所有客户偏好信息，因此，合理使用数据湖能有效指导企业或组织为客户提供个性化服务。

用户在各种数字化平台上留下的大量数据足迹，普通的事务系统难以应付，而这正是数据湖在企业中展示其作用的绝佳时期。

除了来自事务系统的数据之外，还有大量的数据被收集，比如行为数据（主要是客户端脚本），以及来自各种社交媒体渠道的数据。除了行为数据之外，某些设备还可以捕捉位置和手势，这些位置和手势可以被进一步处理，以支持物联网（IoT）用例，如向用户推荐商品或提供产品改进建议。

此外，还需要考虑分析社交媒体中与特定公司相关的舆情信息（针对特定事件的情感分析），并根据一套明确的规则来处理这些内容，如图 12-2 所示。

12.3　提升仓储管理效率

提升仓储管理效率是与业务结合更紧密的用例，在许多决策场景中，数据湖可以非常方便地使用。如果读者所在的公司拥有自己的生产线，或需要处理货运和物流，那么这个用例与读者非常相关，并且适合用于指导数据湖的实现。

在现代仓库中，物联网有各种各样的形式和规模。这些设备可用于仓库中的各种移动设备上，可以在每个产品中进行跟踪和监控。标记这些设备固然很棒，但是要利用这些设备持续产生的海量数据，数据湖是收集这些数据的完美解决方案，之后便可以使用一组定义好的规则来理解这些数据。这可以极大地提高仓库操作的效率，从而增加收入和利润。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

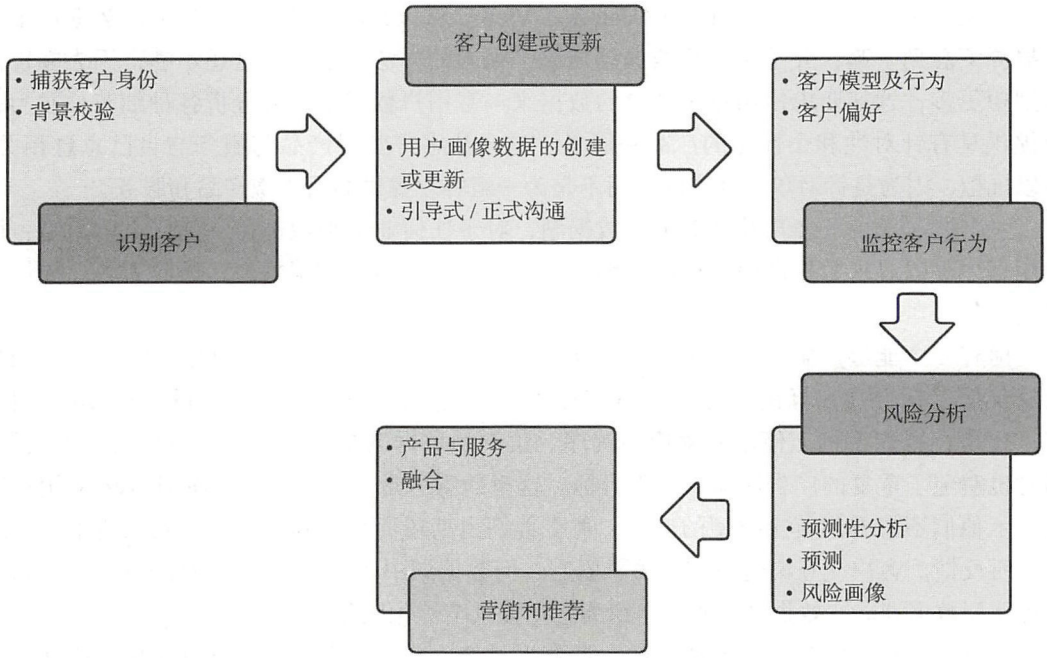


图 12-2 典型的客户了解（Known Your Customer，KYC）过程

如果来自各种事务型应用程序的数据流与物联网设备相连，积极地分析历史数据能优化新订单货物的放置及整个仓库的使用。同样，如果人员管理系统数据也被集成，也可以尝试有效地分配工作，更合理地应用宝贵的时间。

拥有一个数据湖，并对其中数据进行适当的分析，可以帮助企业改进业务，也可以使企业更有信心、更迅速地做出商业决策。这些建议和决定在优化企业服务成本和时间方面发挥着重要作用。图 12-3 展示了仓储管理的流程。

12.4 品牌创建与企业营销

商业环境已经发生了很大的变化，尤其是在最近的 10 年里，现在已经形成为一个由客户驱动的商业社会。客户希望能被理解并得到预期的服务。许多客户愿意分享他们的隐私信息，以便企业能够更好地理解他们，在此之上提供更多有针对性的、为用户定制的服务，以迎合他们在这个数字时代的各种偏好。

客户是上帝，企业要时刻满足用户需求。数据湖中进行的数据分析可以帮助企业更好地了解客户。它可以作为一种挖掘品牌价值的方法，并能以非常合适的方式参与规划产品定位及营销。

数据分析可以帮助企业以一种积极的方式把握市场趋势，帮助企业做出更好的决策，并积极主动地开辟新的业务方向。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

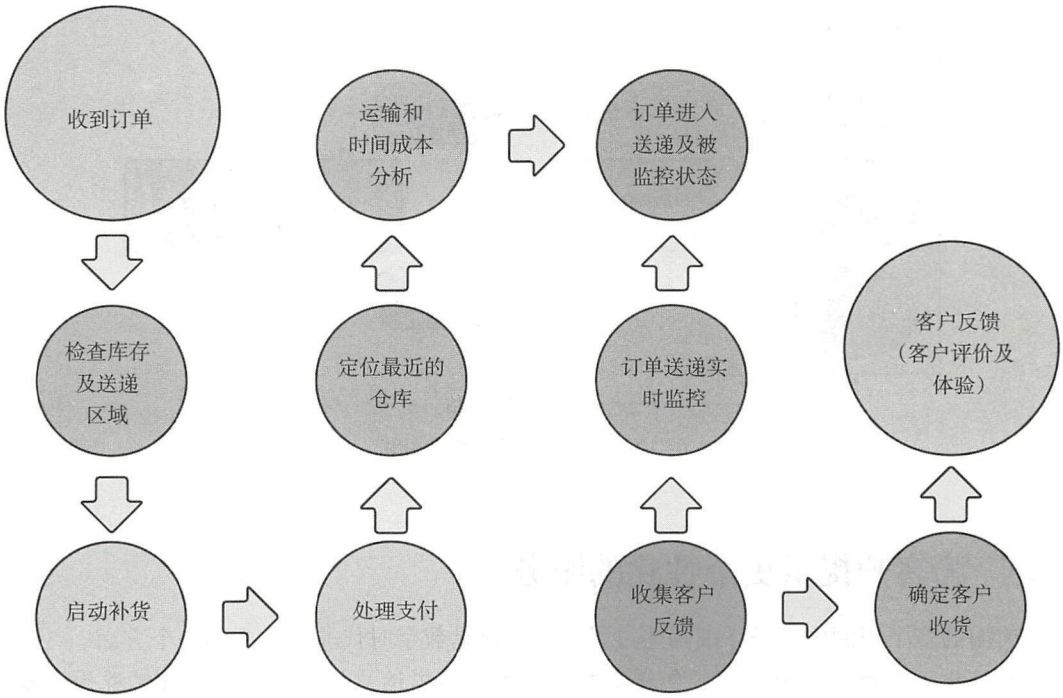


图 12-3　仓储管理

不管你喜欢与否，都将不得不收集来自各种数据源的数据，尤其是公开的数据，用以构建商业决策和分析。只有当我们拥有一个集成了多种可用分析工具的数据湖时，这才是可能的。通常，公开可用的数据的体量非常大，此时数据分析只能依赖大数据技术。

新产品投入市场可能会引领新的消费潮流，数据分析能帮助捕获这种趋势。数据分析能为将企业提升为一个消费者可以信任的品牌提供助力。

前面所述的那些事情，需要大量的研究、营销、广告来支撑。数据湖可以提供完成这些任务所需的某种能力。各种分析可以用来寻找新的市场，刺激产品销售，提升销量及品牌价值。

如今个性化服务的需求旺盛，数据湖可以帮助企业收集大量的客户消息，并可以从对这些消息的分析中学习到有价值的信息，然后基于这些信息向客户推荐合适的商品。

如今，通过社交媒体适时与用户交互来获取其情感倾向是非常关键的。数据湖可以充当侦听器的角色，可以进行一些初始过滤，结合人工智能、机器学习等领域最新技术的使用，确保最重要的客户优先被服务。能有效理解用户情绪并采取应对措施，是产品的一大亮点，能显著提高品牌知名度，使它更有价值。

数据湖不仅可以用来分析自有的商业数据，还可以用来分析竞争对手的公开数据，利用这些分析结果，可对业务流程和商业决策做对应的调整以获取更大的商业利益。

图 12-4 展示了品牌营销与情感分析之间的关系。

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

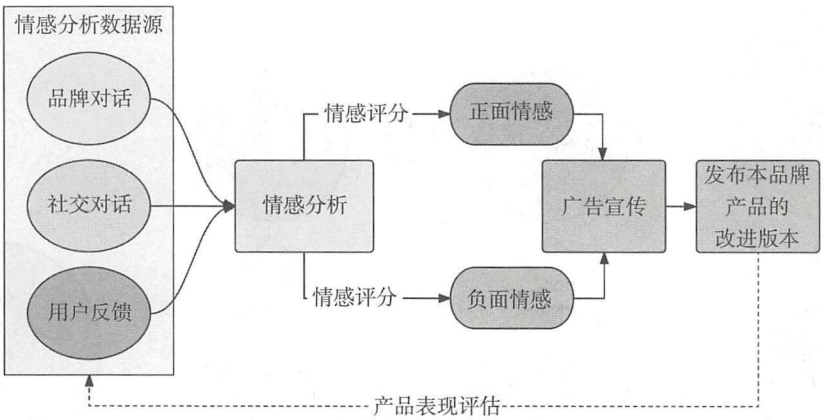


图 12-4 品牌营销与情感分析

12.5 为客户提供更个性化的服务

前面的多个用例中已经详细阐述过了，个性化是数字时代非常重要的一个方面。

个性化，有时也称为定制，包括定制服务或产品以适应特定的个人或特定群体。各种组织借助个性化服务来提高客户满意度、数字销售转化率、营销结果、品牌价值，以及改进的网站运营指标和广告投放。个性化是社交媒体和推荐系统中的关键元素。

——维基百科

上面这个定义清楚地阐释了个性化的含义。在现代社会，客户希望企业通过各种渠道用一致性的方式来了解他们。当客户在计算机或移动设备上浏览网页时，企业需要通过业务系统来理解他们，当客户真正进入商店或组织时，则应通过各种物联网机制来了解他们。

Web 个性化建立在对数据湖中用户行为数据（从用户历史浏览记录和其他客户端收集）和实际交易数据（用户使用业务系统完成的实际的交易数据，如客户详细信息、订单细节、支付细节等）的分析之上。通常，这些交易数据的拥有者并不是数据湖；相反，它是从各种事务源系统传递过来的，并以一种整体的方式进行整合。用户在线浏览时，这些数据被用于提供实际的个性化服务或推荐。普通的事务型应用程序涉及的分析非常有限，而且它们是互相隔离的（它们不与其他应用程序交互，并且不知道其他应用程序是否已经与客户交互过），基于这个原因，事务型应用程序很难对客户有整体的认识。如果客户在其他渠道中已经拒绝了一个特性相似的产品，那么数据湖就帮助企业避免向客户推荐重复的产品。例如，如果在移动应用中，会通过移动应用发起针对某一特定产品的活动，但是客户拒绝该产品或不回应。当客户在网站上与业务系统交互时，这可以作为避免向客户推荐相同商品的一种手段。相反，可以借机向客户推荐另外一件产品，然后观察客户的反馈。只有当存在一个存储空间容纳所有数据，并基于这些数据进行各种分析以后，才能形成一个整体的判断，

仅供非商业用途或交流学习使用

非卖品，仅供非商业用途或交流学习使用

之后这些判断可以用于各种事务系统。

可预见的将来会有越来越多的客户交互渠道，对于每个客户来说，会在这些交互渠道中留下数字足迹，这些信息是极具商业价值的，因此对客户个性化数据的收集刻不容缓。

12.6　让物联网数据触手可及

物联网（Internet of things，IoT）是连接物理设备、车载设备（也称为“联网设备”或“智能设备”）、建筑物及其他嵌入式电子设备、软件、传感器、执行器的网络，设备之间可以通过物联网收集和交换数据。

——维基百科

物联网的世界正在以非常快的速度发展。为了取得全方位的成功，或赶上竞争对手的步伐，每个企业都必须在这个领域投入时间和金钱。物联网会产生大量的数据，数据湖是对各种设备产生的海量数据进行存储和分析的首选工具，分析结果通常具有商业价值。

企业可以将物联网的触角延伸到多处，包括企业内部或外部，根据不同的设置规则，可以在多个有用的业务场景中使用物联网设备产生的数据。例如，可以使用物联网来管理仓库，如维护和管理仓库中的各种资产，跟踪仓库中的各种产品，等等。然而，只有当从物联网设备持续收集数据成为可能时，才能基于数据进行分析，获得有价值的洞察力，用于对商业行为做出反应。

在实体商店中，这些设备可以安装在商店的各个角落，当用户靠近它们或在附近时，可以使用客户数据来启动个性化服务。例如，如果客户安装了商家的移动应用，并且已经在他们的设备上启用了蓝牙功能，当他们走进商店时，移动设备就会开始使用蓝牙与商店里的另一个蓝牙设备进行通信。例如，在商店中，显示设备可以定制和存储更适合附近客户查看的商品，而不会对他们的私有数据进行过多的干扰。还有许多其他的案例，在这些案例中可根据业务需求，使用更多的设备。

12.7　更实用的数据归档

企业中的任何事务系统都将数据保存在生产数据存储系统中，这对数据操作来说是必要的。历史数据是企业及其运营所需的“金矿”。历史数据能为业务运营的变化提供有价值的信息，并为未来业务的运营方向转换提供参考依据。

如今，当不需要历史数据或将其从事务数据存储系统中移除时，这些数据经常被保存在硬磁带中。在硬磁带中保存的数据通常被忽略，闲置在那里毫无用处。数据湖可以用作一个活跃的归档存储系统来替代硬磁带，用于存储被认为是非生产性的数据。这是非常有用的，因为如果数据存储在数据湖中，则更容易恢复为活跃数据，如果存储在硬磁带中，则很难恢复。

非卖品，仅供非商业用途或交流学习使用

12.8 现有的数据仓库基础设施

在大数据时代到来之前，分析工作是在传统的数据仓库中完成的。大数据为分析工作铺平了道路，使其能够以更有效的方式进行分析，并具有很大的灵活性。然而，传统的数据仓库仍然存在，原因有很多，而且不会很快消失。

数据湖可以以各种方式增强和支持数据仓库。数据仓库可以用于生成各种简单报表。业务系统可以在事务数据模型之上应用语义数据仓库模型，然后快速创建报表，这是事务应用程序每天处理的事务。

某些数据仓库的需求可以通过数据湖基础设施来很好地实现，并且由于它们有大量的历史数据，在许多业务场景中都是非常有用的。

12.9 实现电信安全和法规遵从

这更像是一个行业相关的用例，在这种情况下，全国性的电信公司根据法规遵从性（compliance），依照签订的所有安全协议，协助执法机构调查网络犯罪行为，并偶尔进行审计。随着技术进步、社会发展轨迹和网络犯罪分子从事不道德活动方式的多样化，这些规定变得更加严格。这极大地扩展了电信公司的业务范围，从法规遵从性的角度来看，如今执行了更多的实时和渗透驱动检测操作，以及对网络数据的批量分析。可以对数据记录进行深度分析，以便在一段时间内为每个调用者建立一个通信图（见图 12-5），从而便于在需要时可以重放时间序列数据。这些机制对执法机构的调查非常有用。在这里，我们讨论的通过各种电信提供商的网络传输的大量数据，可能会在几个月的时间内累积到 PB 级别。要实现上面讨论的所有功能，唯一的方法是利用大数据功能进行数据收集、处理和分析。

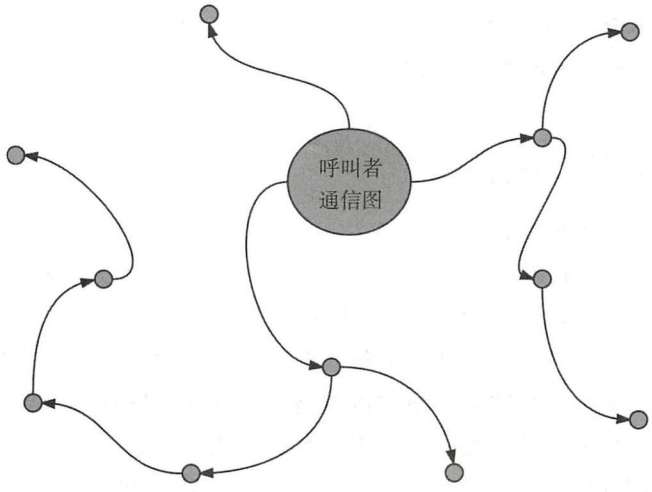


图 12-5 通信图分析

非卖品，仅供非商业用途或交流学习使用

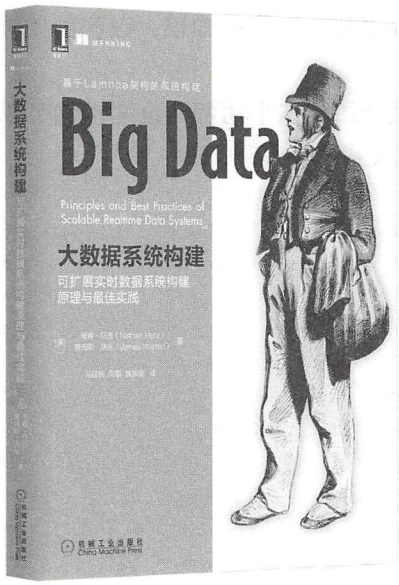
12.10　总结

本章的目的是提供几个数据湖用例供读者思考。这里确保没有将这些用例与特定行业捆定。希望通过这些用例为读者提供足够的知识，以帮助读者为所在企业及其运营活动构建自己的用例。

到这里，我们已经成功地介绍了多个用例，这些用例可以作为读者向所在组织的领导或管理层科普数据湖价值的完美素材。

非卖品，仅供非商业用途或交流学习使用

推荐阅读



大数据系统构建：可扩展实时数据系统构建原理与最佳实践

作者：南森·马茨 等 译者：马延辉 等 书号：978-7-111-55294-9 定价：79.00元

Storm之父，前Twitter的首席工程师撰写，
从系统构建的角度全面、透彻分析企业团队如何更好地利用大数据系统

本书教你使用一种专门设计用来获取和分析网络规模数据的架构去构建大数据系统——Lambda架构，它是一种可扩展的、易于理解的、可以被小团队用来构建和运行大数据系统的方法。除了与你分享Lambda架构的相关知识，本书还给出了相应的示例，将“理论应用于实践”，助你更好地“认识”Lambda架构，更好地将其应用到工作中。

超越了个别工具或平台。任何从事大数据系统工作的人都需要阅读。

—— Jonathan Esterhazy, Groupon

一次全面的、样例驱动的Lambda架构之旅，由Lambda架构的发起人为您指导。

—— Mark Fisher, Pivotal

内含只有在经历许多大数据项目后才能获得的智慧。这是一本必须阅读的书。

—— Pere Ferrera Bertran, Datasalt

非卖品，仅供非商业用途或交流学习使用

作者简介

汤姆斯·约翰（Tomcy John）

是一名企业级Java技术专家，拥有工学学士学位，并且有超过14年多行业的开发经验。他目前担任Emirates Group IT核心架构部门的首席架构师。在此之前，他曾在甲骨文公司、安永公司任职。他主要擅长构建企业级应用，并且在企业内担任首席导师和布道者，以促进新技术融入企业标准技术栈。

潘卡·米斯拉（Pankaj Misra）

是一名技术传播者，拥有工程学士学位，并且有超过16年跨多个业务领域的技术经验。自2015年以来，他一直在Emirates Group IT任职。他擅长架构和构建多技术栈的解决方案及实现。他在印度的技术论坛上也是一名活跃分子，参与过多个海量数据、可水平扩展的近实时数据处理和分析产品的构建。

译者简介

张世武 BBD（数联铭品科技有限公司）北京研发中心负责人，曾先后在中科院计算所、新浪、汽车之家、国美大数据研究院等机构与公司从事搜索引擎研发和管理工作。感兴趣的研究领域包括全文索引内核、分布式搜索引擎框架、相关性排序、机器学习算法等。译著有《深入理解Elasticsearch》《实用机器学习》等。

李想 现就职于阿里巴巴集团蚂蚁金服人工智能部，曾先后就职于百度、蓝汛、BBD等公司，长期专注于大数据计算、流式计算领域。Hadoop生态圈的各项技术，尤其是分布式计算和流式计算，在蚂蚁金服AI团队有着广泛的应用；该团队也对部分底层技术进行了深度定制。

张浩林 BBD高级副总裁，曾先后在华为技术有限公司（任产品经理）和东软集团有限公司（任财税事业部总经理）工作，从事光网络通信技术、机器学习与人工智能解决方案、大数据行业应用等方面的研究与应用工作。

印象中这是我读过的第一本数据湖著作。数据湖逐渐演变成大数据架构的通用解决方案。本书对数据湖及Lambda架构有非常精准的介绍，理论性跟实践性结合得很好。在WiFi万能钥匙的很多业务线中都有数据湖应用场景，如果能早点读到本书，相信能极大减少技术探索的时间。

——章动，WiFi万能钥匙精准内容大数据总监

数据湖遵循了Lambda架构设计理念，整合了现有的各项大数据技术，从而实现了离线与在线、批量与流式的有机统一。本书介绍的设计思想和具体技术实现，包括对现有技术的评估选型，都极具参考价值。

——梁磊，蚂蚁金服人工智能高级技术专家

数据湖技术涵盖了大数据处理的全周期。本书对4V、数据获取、消息队列、ETL、大数据存储、计算引擎、Lambda架构等方面有非常精彩的叙述，深入浅出，图文并茂，值得各种层次的读者研读。

——左春伟，淘宝高级技术专家

如今，大数据技术通常涉及多种框架的整合，初学者往往深陷技术细节的泥潭难以自拔。数据湖是对大数据技术的系统性梳理，对其进行了模块化、可插拔的架构设计，能满足4V的弹性要求。本书对数据湖原理及Lambda架构有着非常详细的阐述，值得推荐。

——陈铁兵，滴滴出行资深工程师

数据湖技术是基于现有大数据技术深度整合出来的一套全面的解决方案。在DT时代，企业在运营决策上对数据有强烈的依赖，数据湖技术不仅是技术框架，更是一种思维框架。本书对数据湖技术及其设计思维方式进行了全面而深入的介绍，强烈推荐给数据开发相关的读者。

——王尧，美团大众点评技术专家

[PACKT] PUBLISHING

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

上架指导：大数据
ISBN 978-7-111-61553-8
9 787111 615538
定价：99.00元